

版权注意事项：1、书籍版权归著者和出版社所有；  
2、本PDF仅用于个人获取知识，进行私底下知识交流；  
3、PDF获得者不得在互联网以任何目的进行传播；  
如有需要，请尽量购买正版实体书！支持书籍作者！！



- OpenStack是一个云管理操作系统，用来控制数据中心中的计算、存储、网络资源池
- 管理员通过一个Web界面为用户提供所需的资源

# 每天5分钟

Open Source Cloud Computing Platform

# 玩转OpenStack

CloudMan 编著

Big Data

Cloud Computing

Internet of Things

清华大学出版社



# 每天5分钟玩转

## OpenStack

CloudMan 编著

清华大学出版社  
北京

## 内 容 简 介

本书是一本 OpenStack 的教程和参考。读者在学习的过程中,可以跟着教程进行操作,在实践中掌握 OpenStack 的核心技能。在之后的工作中,则可以将本教程作为参考书,按需查找相关知识点。

本书共分为两大部分。第一部分介绍虚拟化和云计算基础知识,重点讲解 KVM 的理论和实践。第二部分首先介绍 OpenStack 架构,演示如何搭建 OpenStack 环境,然后逐一详细讲解 OpenStack 各个核心模块,包括 Keystone、Glance、Nova、Cinder 和 Neutron。

本书适合 OpenStack 初学者、云计算技术人员、云计算研究人员等使用,也适合高校和培训学校相关专业的师生教学参考。

本书封面贴有清华大学出版社防伪标签,无标签者不得销售

版权所有,侵权必究。侵权举报电话:010-62782989 13701121933

### 图书在版编目(CIP)数据

每天 5 分钟玩转 OpenStack / CloudMan 编著. — 北京:清华大学出版社,2017  
ISBN 978-7-302-45531-8

I. ①每… II. ①C… III. ①计算机网络 IV. ①TP393

中国版本图书馆 CIP 数据核字(2016)第 277403 号

责任编辑:夏毓彦

封面设计:王 翔

责任校对:闫秀华

责任印制:沈 露

出版发行:清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址:北京清华大学学研大厦 A 座 邮 编:100084

社总机:010-62770175

邮 购:010-62786544

投稿与读者服务:010-62776969, [c-service@tup.tsinghua.edu.cn](mailto:c-service@tup.tsinghua.edu.cn)

质 量 反 馈:010-62772015, [zhiliang@tup.tsinghua.edu.cn](mailto:zhiliang@tup.tsinghua.edu.cn)

印 装 者:北京嘉实印刷有限公司

经 销:全国新华书店

开 本:190mm×260mm

印 张:27

字 数:691 千字

版 次:2017 年 1 月第 1 版

印 次:2017 年 1 月第 1 次印刷

印 数:1~3500

定 价:89.00 元

产品编号:070923-01

# 前言

## 写在最前面

这是一个 OpenStack 教程，有下面两个特点：

- 系统讲解 OpenStack。从架构到各个组件；从整体到细节逐一讨论。
- 重实践并兼顾理论。主要从实际操作的角度带着大家学习 OpenStack。

## 为啥要写这个

简单回答是：因为 OpenStack 学习难度大，但如果掌握了，价值会很大。

先做一个自我介绍吧。

本人网名 CloudMan，在 IT 这个行当已经摸爬滚打了十多年，2005 年之前是搞上层应用开发的，那时候 Java 比较火，所以 J2EE 相关的技术搞得比较多。后来入职一家大型 IT 公司，公司的产品从中间件到操作系统，从服务器到存储，从虚拟化到云计算都有涉及。

本人所在的部门是专门做 IT 基础设施实施服务的，项目涉及服务器、存储、网络、虚拟化、云各个方面，而且这个部门的重要任务是为公司 IT 市场最新和最热门的领域开疆扩土。比如前几年的虚拟化，这两年的云计算和大数据。

可以说部门的这个定位非常符合我的技术偏好。我对新技术长期保持着浓厚的兴趣和学习热情，所以在这个部门一待就是十几年，而且一直搞技术，虽然现在的头衔是架构师，平时还是一直坚持实际动手操作，否则会没有安全感。

好，现在回到 OpenStack 这个主题。

本人是在 2013 年开始接触 OpenStack，虽然具备比较扎实的技术功底，在经过一段时间的学习后，还是感觉 OpenStack 这个东西上手不太容易，个人认为有以下几个原因：

### 1. OpenStack 涉及的知识领域极广

可以说涵盖了 IT 基础设施的所有范围，计算、存储、网络、虚拟化、高可用、安全、灾备无所不包，即便是像我这种每天都在这个领域工作的人也感觉压力颇大。

### 2. OpenStack 是一个平台，不是一个具体的实施方案

OpenStack 的各个组件都采用 Driver 的架构，支持各种具体的实现技术。比如 OpenStack 的存储服务 Cinder 只定义了上层抽象 API，具体的实现交给下面的各种 Driver，比如基于 LVM



的 iSCSI Driver, EMC、IBM 等商业存储产品的 Driver, 或者是开源的分布式存储软件, 比如 Ceph、GlusterFS 的 Driver。

正是因为这种架构上的灵活性, 使得初学者在学习 OpenStack 的时候不会像学习其他具体软件产品那样容易上手。

### 3. OpenStack 本身是一个分布式系统

大多数搞 IT 的对分布式计算都不会太熟悉, 直接冲进来会被 OpenStack 繁多的组件以及它们之间的交互方式搞得云里雾里。

虽然 OpenStack 学习曲线比较陡峭, 掌握起来难度较大, 但 OpenStack 目前已经是 IaaS 云的事实标准, 而且前途一片光明, 对于我们搞 IT 的如果能啃下这个骨头, 必定能大大提升自身的竞争力。

## 写给谁看

这套教程的目标读者包括:

### 1. OpenStack 初学者

我学习 OpenStack 也是经历了一个艰辛曲折的过程, 其主要原因在于没有找到一个系统讲解 OpenStack 的教程, 大部分资料都比较分散, 对于初学者无法有机地串起来。也正是因为这个原因, 让我萌发了编写这样一套教程的想法, 能够让初学者少走弯路, 系统地学习、掌握和实践 OpenStack。

### 2. OpenStack 实施工程师

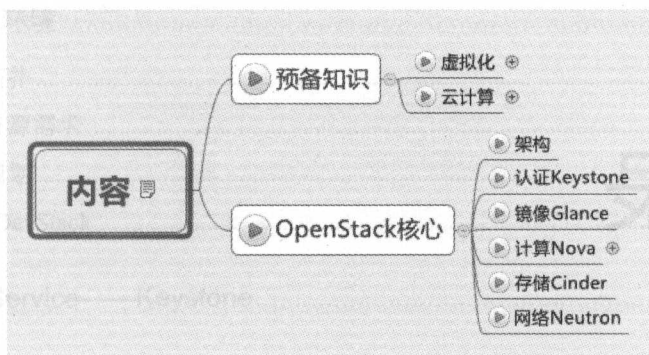
之前说了, 我在公司的职位是架构师, 但骨子里我更把自己定位成一位能到一线攻城拔寨的实施工程师。所以这个教程也是针对 OpenStack 的实施人员, 让他们能够通过学习真正掌握部署 OpenStack 的知识、技能以及故障排查技巧。

### 3. 我自己

写这个教程同时也是对自己这几年学习和实践 OpenStack 的一个总结。我觉得: 对于知识, 只有把它写出来并能够让其他人理解才能真正说明自己掌握了这项知识。

## 包含哪些内容

本书两大块内容, 如下图所示。



## 1. 预备知识

因为面向初学者，首先会有虚拟化和云计算的“预备知识”，会介绍 KVM、IaaS 等技术。

## 2. OpenStack 核心

这是主要内容，包含 OpenStack 的架构和各个核心组件。将会通过大量的案例、操作步骤、截图、日志来帮助大家理解 OpenStack 各组件是如何工作的。其目标是让各位可以根据客户的需求进行配置和调整。

## 怎样的编写方式

在当下这个共享经济时代，我觉得应该用互联网的方式来分享知识和心得。这个教程会通过我的微信公众号（cloudman6）每周一、周三、周五定期发布。

用公众号我觉得有两个好处：

- （1）可以随时随地查看和浏览已推送的内容。
- （2）可以通过公众号跟我互动，提出问题和建议。

## 为啥叫《每天 5 分钟玩转 OpenStack》

为了降低学习的难度并且考虑到移动端碎片化阅读的特点，每次推送的内容大家只需要花 5 分钟就能看完（注意，这里说的是看完，有时候要完全理解可能需要更多时间哈），每次的内容只包含 1~3 个知识点，这也是我把教程命名为《每天 5 分钟玩转 OpenStack》的原因。虽然是碎片化推送，但整个教程是系统、连贯和完整的，只是化整为零了。

好了，今天这 5 分钟算是开了个头，下面我们正式开始玩转 OpenStack。

编者

2016 年 10 月

# 目 录

## 第一篇 预备知识

第 1 章 虚拟化 .....	2
1.1 1 型虚拟化 .....	2
1.2 2 型虚拟化 .....	2
1.3 KVM .....	3
1.3.1 基本概念 .....	3
1.3.2 KVM 实操 .....	4
1.4 KVM 虚拟化原理 .....	11
1.4.1 CPU 虚拟化 .....	11
1.4.2 内存虚拟化 .....	12
1.4.3 存储虚拟化 .....	13
1.5 网络虚拟化 .....	19
1.5.1 Linux Bridge .....	19
1.5.2 VLAN .....	28
1.5.3 Linux Bridge + VLAN = 虚拟交换机 .....	35
第 2 章 云计算 .....	36
2.1 基本概念 .....	36
2.2 云计算和 OpenStack .....	38

## 第二篇 OpenStack 核心

第 3 章 OpenStack 架构 .....	41
3.1 Conceptual Architecture .....	41
3.2 Logical Architecture .....	42

第 4 章 搭建实验环境 .....	45
4.1 部署拓扑 .....	45
4.2 物理资源需求 .....	46
4.3 网络规划 .....	47
4.4 部署 DevStack .....	47
第 5 章 Identity Service——Keystone .....	55
5.1 概念 .....	55
5.1.1 User .....	55
5.1.2 Credentials .....	57
5.1.3 Authentication .....	57
5.1.4 Token .....	57
5.1.5 Project .....	58
5.1.6 Service .....	59
5.1.7 Endpoint .....	60
5.1.8 Role .....	60
5.2 通过例子学习 .....	62
5.2.1 第 1 步 登录 .....	62
5.2.2 第 2 步 显示操作界面 .....	62
5.2.3 第 3 步 显示 image 列表 .....	63
5.2.4 Troubleshoot .....	64
第 6 章 Image Service——Glance .....	65
6.1 理解 Image .....	65
6.2 理解 Image Service .....	66
6.3 Glance 架构 .....	66
6.4 Glance 操作 .....	69
6.4.1 创建 image .....	70
6.4.2 删除 image .....	72
6.5 如何使用 OpenStack CLI .....	74
6.6 如何 Troubleshooting .....	77
第 7 章 Compute Service——Nova .....	79
7.1 Nova 架构 .....	80



7.1.1	架构概览 .....	80
7.1.2	物理部署方案 .....	82
7.1.3	从虚机创建流程看 nova-* 子服务如何协同工作 .....	84
7.1.4	OpenStack 通用设计思路 .....	85
7.2	Nova 组件详解 .....	88
7.2.1	nova-api .....	88
7.2.2	nova-scheduler .....	90
7.2.3	nova-compute .....	97
7.2.4	nova-conductor .....	104
7.3	通过场景学习 Nova .....	105
7.3.1	看懂 OpenStack 日志 .....	105
7.3.2	Launch .....	108
7.3.3	Shut Off .....	108
7.3.4	Start .....	112
7.3.5	Soft/Hard Reboot .....	114
7.3.6	Lock/Unlock .....	114
7.3.7	Terminate .....	115
7.3.8	Pause/Resume .....	116
7.3.9	Suspend/Resume .....	118
7.3.10	Rescue/Unrescue .....	119
7.3.11	Snapshot .....	122
7.3.12	Rebuild .....	125
7.3.13	Shelve .....	128
7.3.14	Unshelve .....	130
7.3.15	Migrate .....	133
7.3.16	Resize .....	139
7.3.17	Live Migrate .....	144
7.3.18	Evacuate .....	150
7.3.19	Instance 操作总结 .....	154
7.4	小节 .....	156
<b>第 8 章 Block Storage Service —— Cinder .....</b>		<b>157</b>
8.1	理解 Block Storage .....	157
8.2	理解 Block Storage Service .....	157

8.2.1	Cinder 架构 .....	158
8.2.2	物理部署方案 .....	159
8.2.3	从 volume 创建流程看 cinder-*子服务如何协同工作 .....	160
8.2.4	Cinder 的设计思想 .....	161
8.2.5	Cinder 组件详解 .....	163
8.2.6	通过场景学习 Cinder .....	170
8.3	小节 .....	220
<b>第 9 章 Networking Service ——Neutron</b> .....		<b>221</b>
9.1	Neutron 概述 .....	221
9.1.1	Neutron 功能 .....	221
9.1.2	Neutron 网络基本概念 .....	222
9.2	Neutron 架构 .....	224
9.2.1	物理部署方案 .....	227
9.2.2	Neutron Server .....	228
9.2.3	Neutron 如何支持各种 network provider .....	229
9.2.4	ML2 Core Plugin .....	231
9.2.5	Service Plugin / Agent .....	234
9.2.6	小结 .....	235
9.3	为 Neutron 准备物理基础设施 .....	237
9.3.1	1 控制节点 + 1 计算节点的部署方案 .....	237
9.3.2	配置多个网卡区分不同类型的网络数据 .....	238
9.3.3	网络拓扑 .....	239
9.3.4	安装和配置节点 .....	240
9.4	Linux Bridge 实现 Neutron 网络 .....	244
9.4.1	配置 linux-bridge mechanism driver .....	244
9.4.2	初始网络状态 .....	245
9.4.3	了解 Linux Bridge 环境中的各种网络设备 .....	247
9.4.4	local network .....	248
9.4.5	flat network .....	262
9.4.6	DHCP 服务 .....	270
9.4.7	vlan network .....	274
9.4.8	Routing .....	285
9.4.9	vxlan network .....	307

9.4.10	Securert Group	321
9.4.11	Firewall as a Service	328
9.4.12	Load Balancing as a Service	337
9.5	Open vSwitch 实现 Neutron 网络	358
9.5.1	网络拓扑	358
9.5.2	配置 openvswitch mechanism driver	359
9.5.3	初始网络状态	360
9.5.4	了解 Open vSwitch 环境中的各种网络设备	362
9.5.5	local network	362
9.5.6	flat network	377
9.5.7	vlan network	386
9.5.8	Routing	399
9.5.9	vxlan network	411
9.6	总结	421
	写在最后	422

# 第一篇

---

## 预备知识

OpenStack 是云操作系统,要学习 OpenStack,首先需要掌握一些虚拟化和云计算的相关知识。



# 第 1 章

## ◀ 虚 拟 化 ▶

虚拟化是云计算的基础。简单地说，虚拟化使得在一台物理的服务器上可以跑多台虚拟机，虚拟机共享物理机的 CPU、内存、IO 硬件资源，但逻辑上虚拟机之间是相互隔离的。

物理机我们一般称为宿主机（Host），宿主机上面的虚拟机称为客户机（Guest）。

那么 Host 是如何将自己的硬件资源虚拟化，并提供给 Guest 使用的呢？

这个主要是通过一个叫做 Hypervisor 的程序实现的。

根据 Hypervisor 的实现方式和所处的位置，虚拟化又分为两种：1 型虚拟化和 2 型虚拟化。

### 1.1 1 型虚拟化

Hypervisor 直接安装在物理机上，多个虚拟机在 Hypervisor 上运行。Hypervisor 实现方式一般是一个特殊定制的 Linux 系统。Xen 和 VMWare 的 ESXi 都属于这个类型，如图 1-1 所示。

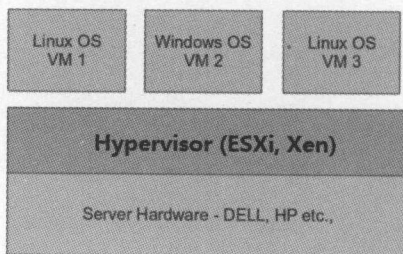


图 1-1

### 1.2 2 型虚拟化

物理机上首先安装常规的操作系统，比如 Redhat、Ubuntu 和 Windows。Hypervisor 作为 OS 上的一个程序模块运行，并对虚拟机进行管理。KVM、VirtualBox 和 VMWare Workstation 都属于这个类型，如图 1-2 所示。

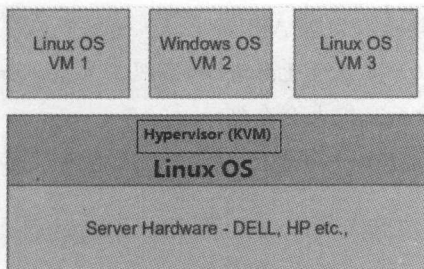


图 1-2

理论上讲：

1. 型虚拟化一般对硬件虚拟化功能进行了特别优化，性能上比 2 型要高；
2. 型虚拟化因为基于普通的操作系统，会比较灵活，比如支持虚拟机嵌套。嵌套意味着可以在 KVM 虚拟机中再运行 KVM。

## 1.3 KVM

下面重点介绍 KVM 这种 2 型虚拟化技术。

### 1.3.1 基本概念

在 x86 平台上最热门、运用最广泛的虚拟化方案莫过于 KVM 了。OpenStack 对 KVM 支持得也最好，我们的教程也理所当然选择 KVM 作为实验环境的 Hypervisor。

KVM 全称是 Kernel-Based Virtual Machine。也就是说 KVM 是基于 Linux 内核实现的。

KVM 有一个内核模块叫 `kvm.ko`，只用于管理虚拟 CPU 和内存。

那 IO 的虚拟化，比如存储和网络设备由谁实现呢？

这个就交给 Linux 内核和 Qemu 来实现。

说白了，作为一个 Hypervisor，KVM 本身只关注虚拟机调度和内存管理这两个方面。IO 外设的任务交给 Linux 内核和 Qemu。

#### Libvirt

大家在网上看 KVM 相关文章的时候肯定经常会看到 Libvirt 这个东西。

Libvirt 是啥？

简单地讲就是 KVM 的管理工具。

其实，Libvirt 除了能管理 KVM 这种 Hypervisor，还能管理 Xen，VirtualBox 等。

OpenStack 底层也使用 Libvirt，所以很有必要学习一下。

Libvirt 包含 3 个东西：后台 daemon 程序 `libvirtd`、API 库和命令行工具 `virsh`。

- libvirtd 是服务程序，接收和处理 API 请求；
- API 库使得其他人可以开发基于 Libvirt 的高级工具，比如 virt-manager，这是个图形化的 KVM 管理工具，后面我们也会介绍；
- virsh 是我们经常要用的 KVM 命令行工具，后面会有使用的示例。

作为 KVM 和 OpenStack 的实施人员，virsh 和 virt-manager 是一定要会用的。

今天 5 分钟差不多了，下一节我们来玩 KVM。

## 1.3.2 KVM 实操

### 1. 准备 KVM 实验环境

上一节说了，KVM 是 2 型虚拟化，是运行在操作系统之上的，所以先要装一个 Linux。Ubuntu、Redhat、CentOS 都可以，这里以 Ubuntu14.04 为例。

基本的 Ubuntu 操作系统装好之后，安装 KVM 需要的包：

```
$ sudo apt-get install qemu-kvm qemu-system libvirt-bin virt-manager
bridge-utils vlan
```

通过这些安装包顺便复习一下上一节介绍的 KVM 的相关知识。

- qemu-kvm 和 qemu-system 是 KVM 和 QEMU 的核心包，提供 CPU、内存和 IO 虚拟化功能。
- libvirt-bin 就是 libvirt，用于管理 KVM 等 Hypervisor。
- virt-manager 是 KVM 图形化管理工具。
- bridge-utils 和 vlan，主要是网络虚拟化需要，KVM 网络虚拟化的实现是基于 linux-bridge 和 VLAN，后面我们会讨论。

Ubuntu 默认不安装图形界面，手工安装一下：

```
sudo apt-get install xinit sudo apt-get install gdm sudo apt-get install
kubuntu-desktop
```

apt 默认会到官网上去下载安装包，速度很慢，我们可以使用国内的镜像站点。

配置/etc/apt/sources.list:

```
deb http://mirrors.163.com/ubuntu/ trusty main restricted universe
multiverse
deb http://mirrors.163.com/ubuntu/ trusty-security main restricted
universe multiverse
deb http://mirrors.163.com/ubuntu/ trusty-updates main restricted
universe multiverse
deb http://mirrors.163.com/ubuntu/ trusty-proposed main restricted
universe multiverse
```



```

deb http://mirrors.163.com/ubuntu/ trusty-backports main restricted
universe multiverse
deb-src http://mirrors.163.com/ubuntu/ trusty main restricted universe
multiverse
deb-src http://mirrors.163.com/ubuntu/ trusty-security main restricted
universe multiverse
deb-src http://mirrors.163.com/ubuntu/ trusty-updates main restricted
universe multiverse
deb-src http://mirrors.163.com/ubuntu/ trusty-proposed main restricted
universe multiverse
deb-src http://mirrors.163.com/ubuntu/ trusty-backports main restricted
universe multiverse

```

然后执行下面命令更新安装包 index:

```
# apt update
```

Redhat 和 CentOS 安装相对简单, 安装过程中选择虚拟化和图形组件就可以了。

小窍门: Ubuntu 默认是不允许 root 通过 ssh 直接登录的, 可以修改 /etc/ssh/sshd\_config, 设置:

```
PermitRootLogin yes
```

然后重启 ssh 服务即可:

```
# service ssh restart ssh
stop/waiting ssh start/running, process 27639
```

在虚拟机上做实验

作为 2 型虚拟化的 KVM, 支持虚拟化嵌套, 这使得我们可以在虚拟机中实验 KVM。

比如我在 VMWare Workstation 中安装了一个 Ubuntu14.04 的虚拟机, 为了让 KVM 能创建。

嵌套的虚拟机, 要把 CPU 的虚拟化功能打开。如图 1-3 所示, 在 VMWare 中设置以下 CPU 的模式。

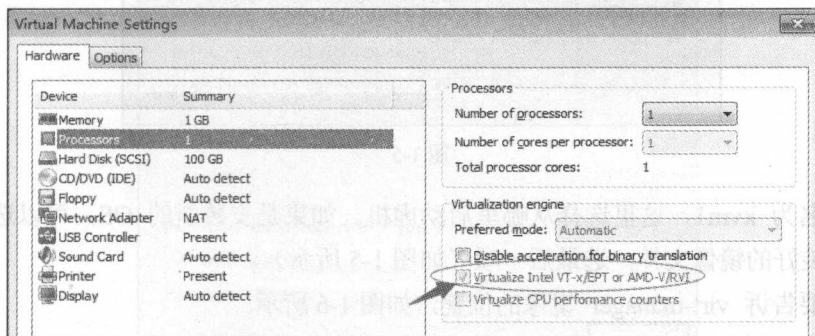


图 1-3

Ubuntu 启动后, 用以下命令确认 CPU 支持虚拟化:



```
# egrep -o '(vmx|svm)' /proc/cpuinfo
# vmx
```

确认 LibvirtD 服务已经启动:

```
# service libvirt-bin status
libvirt-bin start/running, process 1478
```

## 2. 启动第一个 KVM 虚拟机

本节演示如何使用 virt-manager 启动 KVM 虚拟机。

首先通过命令 virt-manager 启动图形界面, 如图 1-4 所示。

```
# virt-manager
```

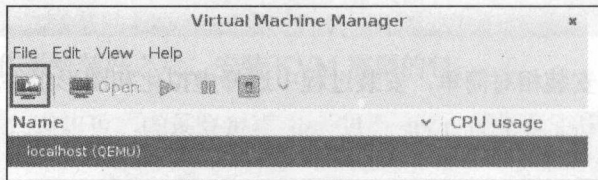


图 1-4

单击图 1-4 中框选的图标创建虚拟机, 如图 1-5 所示。

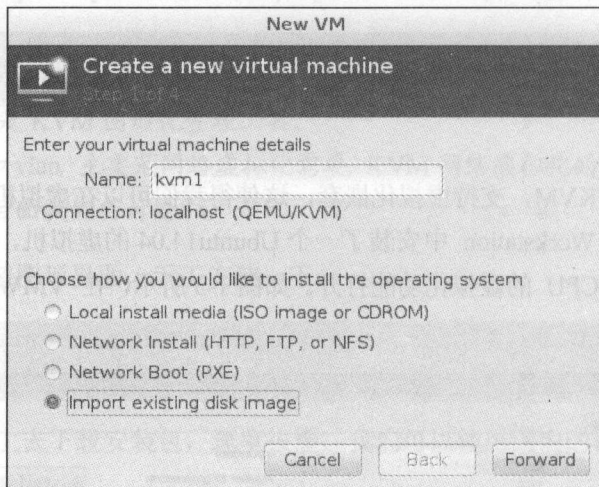


图 1-5

给虚拟机命名为 kvm1, 这里选择从哪里启动虚拟机。如果是安装新的 OS, 可以选择第一项。如果已经有安装好的镜像文件, 选最后一项 (如图 1-5 所示)。

接下来需要告诉 virt-manager 镜像的位置, 如图 1-6 所示。

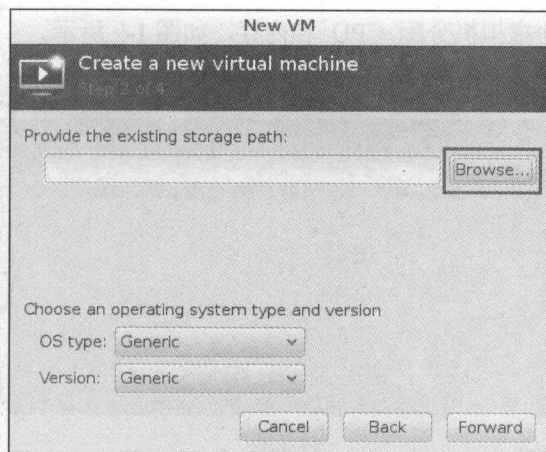


图 1-6

单击“Browser”，打开如图 1-7 所示的界面。

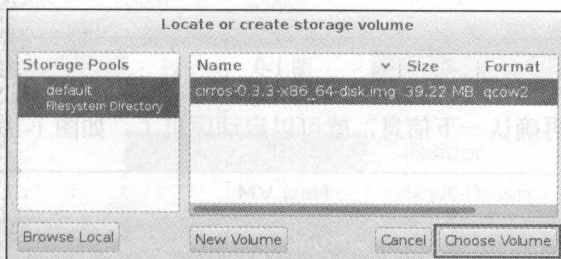


图 1-7

在我的系统中存放了一个 `cirros-0.3.3-x86_64-disk.img` 镜像文件，单击“Choose Volume”，如图 1-8 所示。`cirros` 是一个很小的 linux 镜像，非常适合测试用，大家可以到 <http://download.cirros-cloud.net/> 下载，然后放到 `/var/lib/libvirt/images/` 目录下，这是 KVM 默认查找镜像文件的地方。

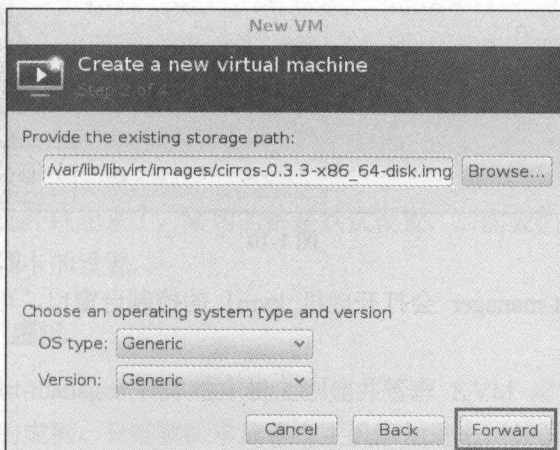


图 1-8

单击“Forward”，为虚拟机分配 CPU 和内存，如图 1-9 所示。

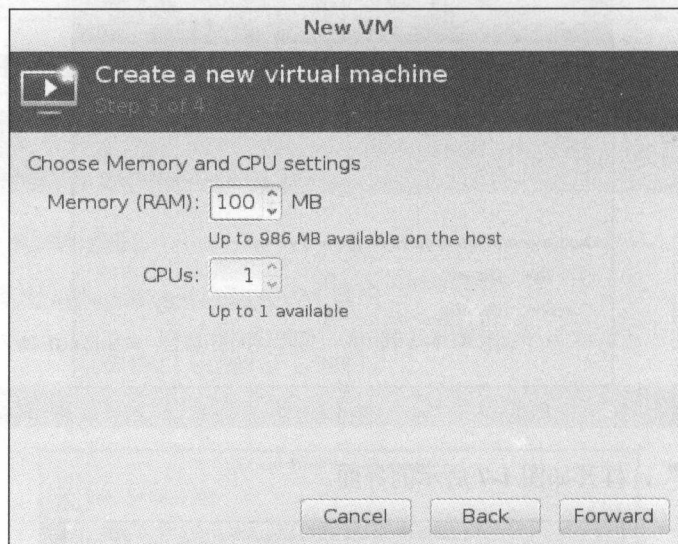


图 1-9

单击“Forward”，再确认一下信息，就可以启动虚拟机了，如图 1-10 所示。

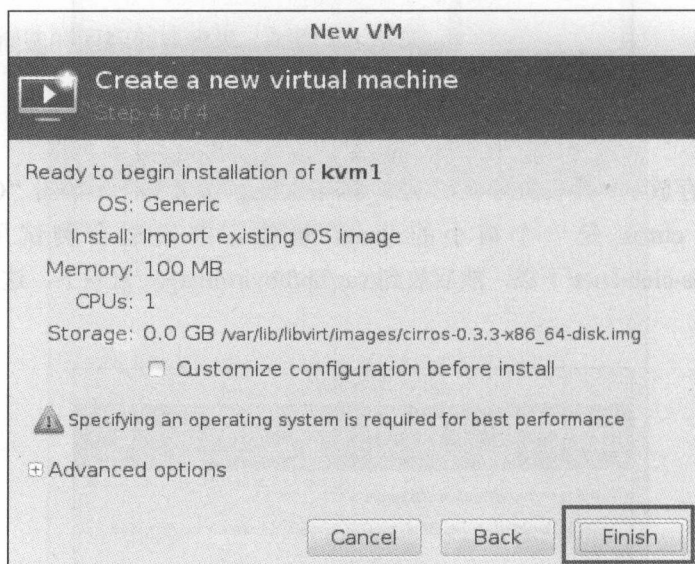


图 1-10

单击“Finish”，virt-manager 会打开虚拟机 kvm1 的控制台窗口，可以看到启动情况，如图 1-11 所示。

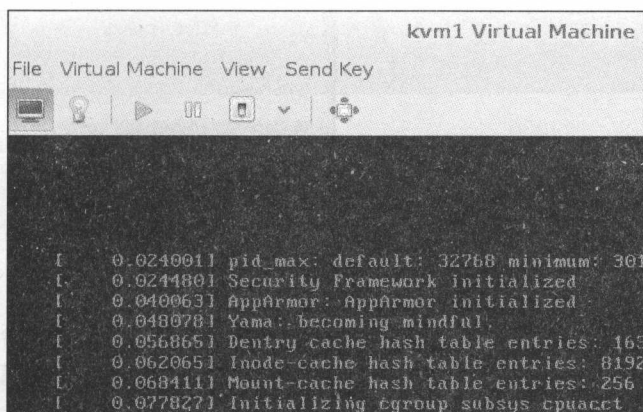


图 1-11

`virt-manager` 可以对虚拟机进行各种管理操作，界面直观友好，很容易上手，如图 1-12 所示。

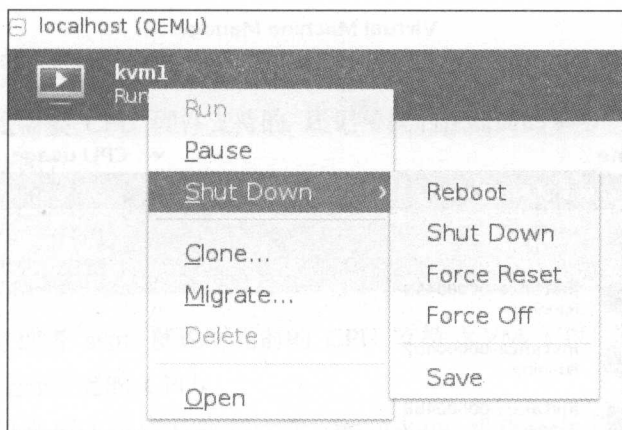


图 1-12

同时我们也可以用命令 `virsh` 管理虚拟机，比如查看宿主机上的虚拟机：

```
root@ubuntu:~# virsh list
Id Name State
-----
 8 kvm1 running
```

至此，第一个虚拟机已经跑起来了，采用的都是默认设置，后面我们会逐步讨论有关虚拟机更细节的内容，比如存储和网卡的设置。

### 3. 远程管理 KVM 虚拟机

上一节我们通过 `virt-manager` 在本地主机上创建并管理 KVM 虚拟机。其实 `virt-manager` 也可以管理其他宿主机上的虚拟机。只需要简单地将宿主机添加进来，如图 1-13 所示。

填入宿主机的相关信息，确定即可，如图 1-14 所示。



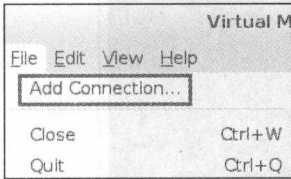


图 1-13

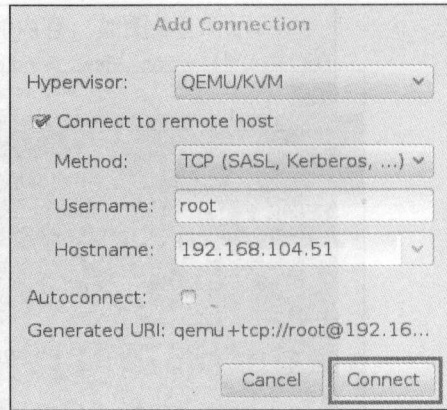


图 1-14

接下来，我们就可以像管理本地虚机一样去管理远程宿主主机上的虚机了，如图 1-15 所示。

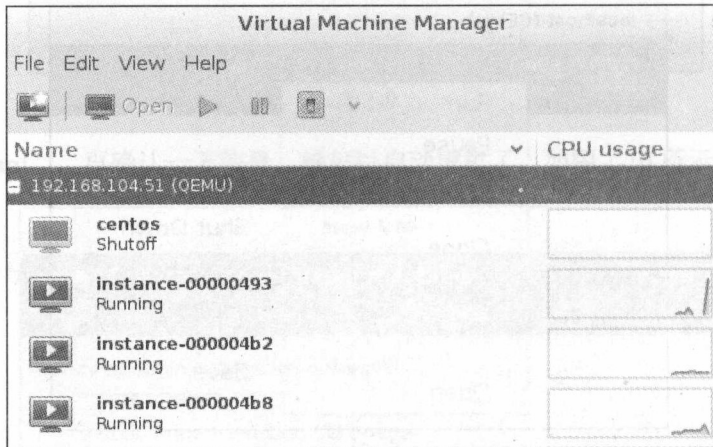


图 1-15

这里其实有一个要配置的地方。

因为 KVM（准确说是 Libvirt）默认不接受远程管理，需要按下面的内容配置被管理宿主主机中的两个文件：

```
/etc/default/libvirt-bin
start_libvirtd="yes"
libvirtd_opts="-d -l"
/etc/libvirt/libvirtd.conf
listen_tls = 0
listen_tcp = 1
unix_sock_group = "libvirtd"
unix_sock_ro_perms = "0777"
unix_sock_rw_perms = "0770"
auth_unix_ro = "none"
```

```
auth_unix_rw = "none"
auth_tcp = "none"
```

然后重启 Libvirt 服务就可以远程管理了。

```
service libvirt-bin restart
```

## 1.4 KVM 虚拟化原理

前面我们成功地把 KVM 跑起来了，有了些感性认识，这个对于初学者非常重要。不过还不够，我们多少得了解一些 KVM 的实现机制，这对以后的工作会有帮助。

### 1.4.1 CPU 虚拟化

KVM 的虚拟化是需要 CPU 硬件支持的。还记得我们在前面的章节讲过用命令来查看 CPU 是否支持 KVM 虚拟化吗？

```
root@ubuntu:~# egrep -o '(vmx|svm)' /proc/cpuinfo
vmx
```

如果有输出 vmx 或者 svm，就说明当前的 CPU 支持 KVM。CPU 厂商 Intel 和 AMD 都支持虚拟化了，除非是非常老的 CPU。

一个 KVM 虚机在宿主机中其实是一个 qemu-kvm 进程，与其他 Linux 进程一样被调度。

比如，在我的实验机上运行的虚机 kvm1 在宿主机中 ps 能看到相应的进程，如图 1-16 所示。

```
root@ubuntu:~# virsh list
 Id      Name                                State
-----
 2       kvm1                                running

root@ubuntu:~# ps -elfgrep kvm1
6.5 libvirt+ 2434      1  3  80      0 - 139654 poll_s 11:22 ?        00:00:44 qemu-system-x86_64 -enable-kvm -name kvm1 -s
100 -realtime mlock-off -smp 1,sockets=1,cores=1,threads=1 -uid 200696f5-f404-0f2a-4087-af52324ab9a9 -no-user-config -n
var/lib/libvirt/qemu/kvm1.monitor server,nowait -mon chardev=charmonitor,id=monitor,mode=control -rtc base=utc -no-shutd
sb,bus=pci.0,addr=0x1.0x2 -drive file=/var/lib/libvirt/images/cirros-0.3.3-x86_64-disk.img,if=none,id=drive-ide0-0-0,for
-drive ide0-0-0,id=ide0-0-0,bootindex=1 -netdev tap,fid=24,id=hostnet0 -device rtl8139,netdev=hostnet0,id=net0,mac=52:54:
charserial0 -device isa-serial,chardev=charserial0,id=serial0 -vnc 127.0.0.1:0 -device cirrus-vga,id=video0,bus=pci.0,ad
dr=0x4 -device hda-duplex,id=sound0-codec0,bus=sound0.0,cad=0 -device virtio-balloon-pci,id=balloon0,bus=pci.0,addr=0x5
```

图 1-16

虚机中的每一个虚拟 vCPU 则对应 qemu-kvm 进程中的一个线程，如图 1-17 所示。

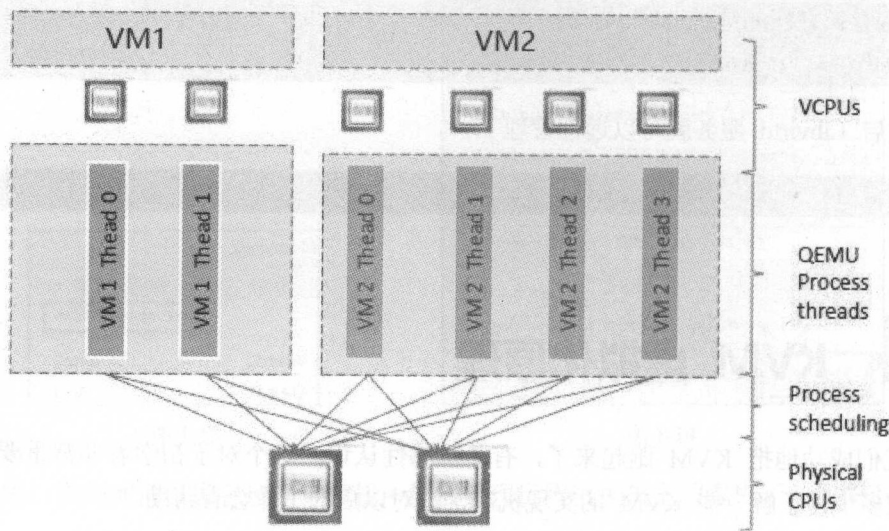


图 1-17

在这个例子中，宿主机有两个物理 CPU，上面起了两个虚机 VM1 和 VM2。

VM1 有两个 vCPU，VM2 有 4 个 vCPU。可以看到 VM1 和 VM2 分别有两个和 4 个线程在两个物理 CPU 上调度。

这里也演示了另一个知识点，即虚机的 vCPU 总数可以超过物理 CPU 数量，这个叫 CPU overcommit（超配）。

KVM 允许 overcommit，这个特性使得虚机能够充分利用宿主机的 CPU 资源，但前提是在同一时刻，不是所有的虚机都满负荷运行。

当然，如果每个虚机都很忙，反而会影响整体性能，所以在使用 overcommit 的时候，需要对虚机的负载情况有所了解，需要测试。

1.4.2 内存虚拟化

KVM 通过内存虚拟化共享物理系统内存，动态分配给虚拟机，如图 1-18 所示。

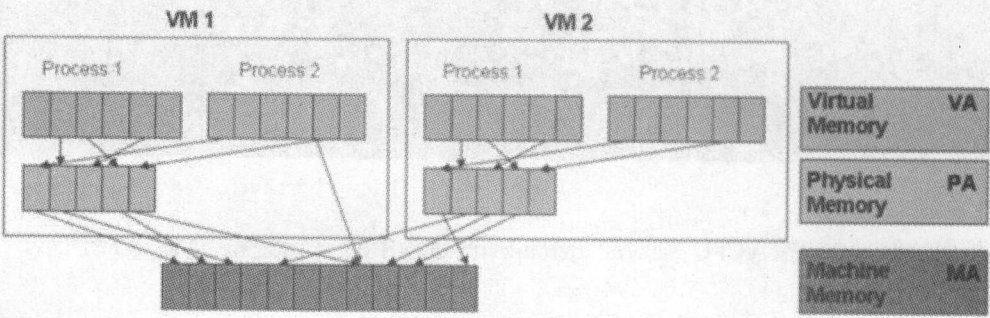


图 1-18

为了在一台机器上运行多个虚拟机，KVM 需要实现 VA（虚拟内存）→ PA（物理内存）→

MA（机器内存）之间的地址转换。虚机 OS 控制虚拟地址到客户内存物理地址的映射（VA → PA），但是虚机 OS 不能直接访问实际机器内存，因此 KVM 需要负责映射客户物理内存到实际机器内存（PA → MA）。具体的实现就不做过多介绍了，大家有兴趣可以查查资料。

还有一点提醒大家，内存也是可以 overcommit 的，即所有虚机的内存之和可以超过宿主机的物理内存。但使用时也需要充分测试，否则性能会受影响。

### 1.4.3 存储虚拟化

KVM 的存储虚拟化是通过存储池（Storage Pool）和卷（Volume）来管理的。

Storage Pool 是宿主机上可以看到的一片存储空间，可以是多种类型，后面会详细讨论。

Volume 是在 Storage Pool 中划分出的一块空间，宿主机将 Volume 分配给虚拟机，Volume 在虚拟机中看到的就是一片硬盘。

下面我们学习不同类型的 Storage Pool。

#### 1. 目录类型的 Storage Pool

文件目录是最常用的 Storage Pool 类型。

KVM 将宿主机目录 `/var/lib/libvirt/images/` 作为默认的 Storage Pool。

那么 Volume 是什么呢？

答案就是该目录下面的文件了，一个文件就是一个 Volume。

大家是否还记得我们之前创建第一个虚机 `kvml` 的时候，就是将镜像文件 `cirros-0.3.3-x86_64-disk.img` 放到了这个目录下。文件 `cirros-0.3.3-x86_64-disk.img` 也就是 Volume，对于 `kvml` 来说，就是它的启动磁盘了。如图 1-19 所示。

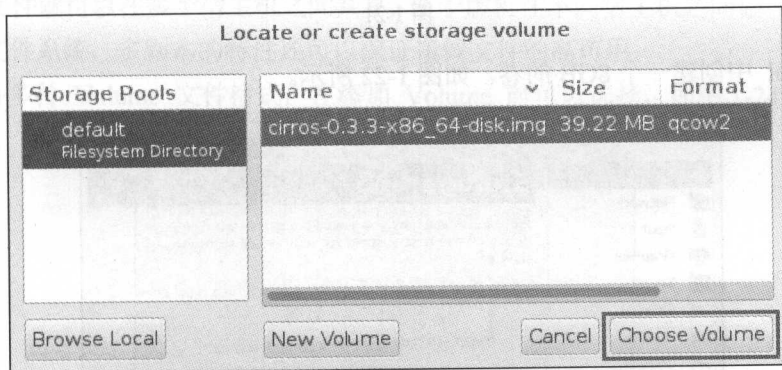


图 1-19

那 KVM 是怎么知道要把 `/var/lib/libvirt/images` 目录当作默认 Storage Pool 的呢？

实际上 KVM 所有可以使用的 Storage Pool 都定义在宿主机的 `/etc/libvirt/storage` 目录下，每个 Pool 一个 xml 文件，默认有一个 `default.xml`，其内容如图 1-20 所示。



```
<pool type="dir">
  <name>default</name>
  <uuid>a78ef0be-4ec8-9929-d5d2-3cd4469bec5d</uuid>
  <capacity unit="bytes">0</capacity>
  <allocation unit="bytes">0</allocation>
  <available unit="bytes">0</available>
  <source>
  </source>
  <target>
    <path>/var/lib/libvirt/images</path>
    <permissions>
      <mode>0711</mode>
      <owner>-1</owner>
      <group>-1</group>
    </permissions>
  </target>
</pool>
```

图 1-20

注意：Storage Pool 的类型是“dir”，目录的路径就是 /var/lib/libvirt/images。  
下面我们为虚机 kvm1 添加一个新的磁盘，看看有什么变化。  
在 virt-manager 中打开 kvm1 的配置页面，右键添加新硬件，如图 1-21 所示。

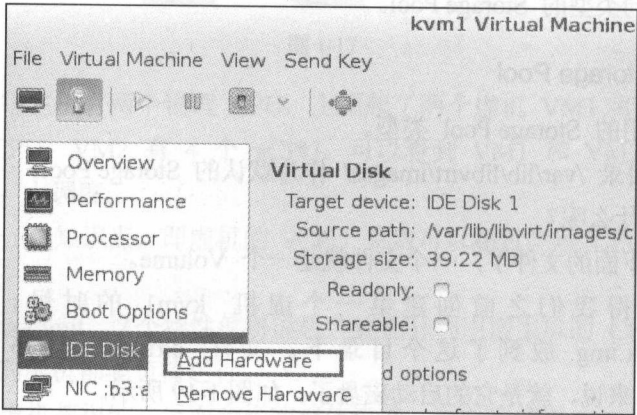


图 1-21

在默认 Pool 中创建一个 8GB 的卷，如图 1-22 所示。

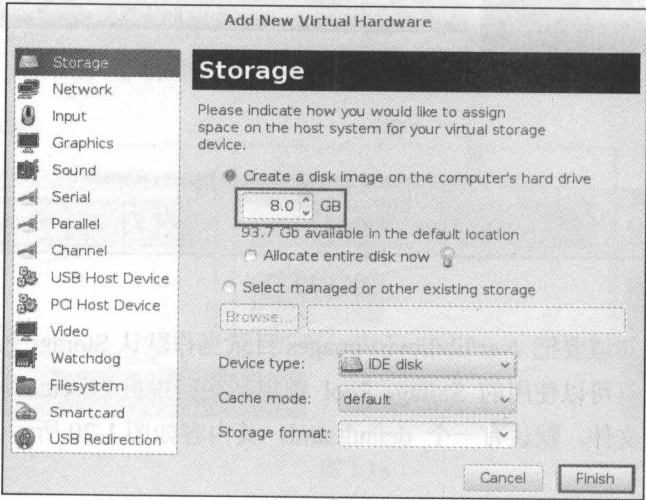


图 1-22

单击“Finish”，可以看到新磁盘的信息，如图 1-23 所示。

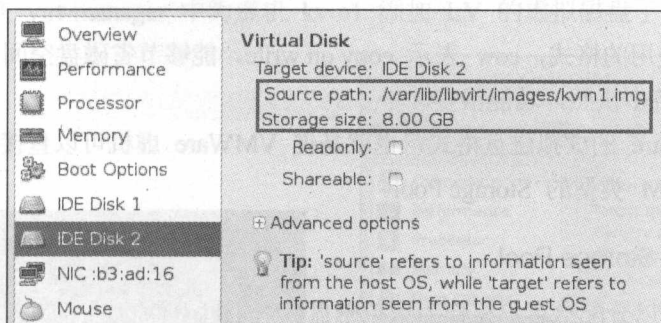


图 1-23

在 `/var/lib/libvirt/images/` 下多了一个 8GB 的文件 `kvm1.img`：

```
root@ubuntu:~# ls -l /var/lib/libvirt/images/
total 14044
-rw-r--r-- 1 root root 14417920 Sep  4 11:24
cirros-0.3.3-x86_64-disk.img
-rw----- 1 root root 8589934592 Sep  4 21:39 kvm1.img
```

使用文件做 Volume 有很多优点：存储方便、移植性好、可复制、可远程访问。

前面几个优点都很好理解，这里对“可远程访问”多解释一下。

远程访问的意思是镜像文件不一定都放置到宿主机本地文件系统中，也可以存储在通过网络连接的远程文件系统，比如 NFS，或者是分布式文件系统中，比如 GlusterFS。

这样镜像文件就可以在多个宿主机之间共享，便于虚机在不同宿主机之间做 Live Migration；如果是分布式文件系统，多副本的特性还可以保证镜像文件的高可用。

KVM 支持多种 Volume 文件格式，在添加 Volume 时可以选择，如图 1-24 所示。

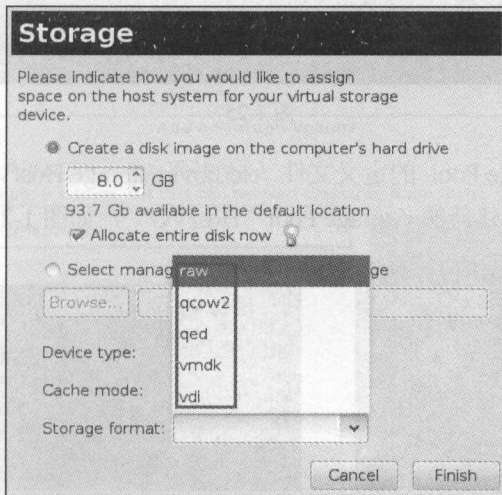


图 1-24

raw 是默认格式，即原始磁盘镜像格式，移植性好，性能好，但大小固定，不能节省磁盘空间。

qcow2 是推荐使用的格式，cow 表示 copy on write，能够节省磁盘空间，支持 AES 加密，支持 zlib 压缩，支持多快照，功能很多。

vmdk 是 VMWare 的虚拟磁盘格式，也就是说 VMWare 虚机可以直接在 KVM 上运行。下一节介绍 LVM 类型的 Storage Pool。

## 2. LVM 类型的 Storage Pool

不仅一个文件可以分配给客户机作为虚拟磁盘，宿主机上 VG 中的 LV 也可以作为虚拟磁盘分配给虚拟机使用。

不过，LV 由于没有磁盘的 MBR 引导记录，不能作为虚拟机的启动盘，只能作为数据盘使用。

这种配置下，宿主机上的 VG 就是一个 Storage Pool，VG 中的 LV 就是 Volume。

LV 的优点是有较好的性能；不足的地方是管理和移动性方面不如镜像文件，而且不能通过网络远程使用。

下面举个例子。

首先，在宿主机上创建了一个容量为 10GB 的 VG，命名为 HostVG，如图 1-25 所示。

```
# vgdisplay
--- volume group ---
VG Name                HostVG
System ID               lvm2
Format                 1
Metadata Areas         1
Metadata Sequence No   2
VG Access               read/write
VG Status               resizable
MAX LV                 0
Cur LV                 1
Open LV                 0
Max PV                 0
Cur PV                 1
Act PV                 1
VG Size                10.00 GiB
PE Size                4.00 MiB
Total PE                2559
Alloc PE / Size        25 / 100.00 MiB
Free PE / Size          2534 / 9.90 GiB
VG UUID                jf301r-jB84-pxmR-MeS7-WVPB-F4uA-8QHm1X
```

图 1-25

然后创建了一个 Storage Pool 的定义文件 /etc/libvirt/storage/HostVG.xml，如图 1-26 所示。

然后通过 virsh 命令创建新的 Storage Pool “HostVG”，如图 1-27 所示。

```
<pool type='lvm'>
  <name>HostVG</name>
  <source>
    <name>HostVG</name>
    <format type='lvm' />
  </source>
  <target>
    <path>/dev/HostVG</path>
  </target>
</pool>
```

图 1-26

```
# virsh pool-list --all
Name      State    Autostart
-----
default   active   yes

# virsh pool-define /etc/libvirt/storage/HostVG.xml
Pool HostVG defined from /etc/libvirt/storage/HostVG.xml

# virsh pool-list --all
Name      State    Autostart
-----
default   active   yes
HostVG    inactive no
```

图 1-27

并启用这个 HostVG，如图 1-28 所示。

现在我们可以 在 virt-manager 中为虚拟机 kvm1 添加 LV 的虚拟磁盘了。如图 1-29 所示。



图 1-28

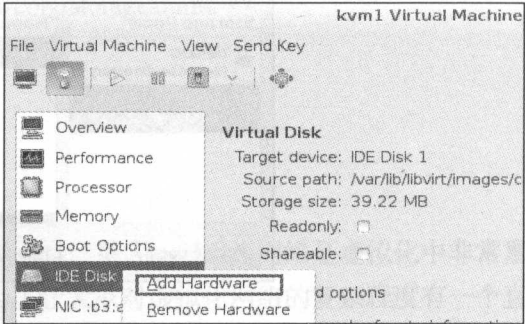


图 1-29

单击“Browse”，如图 1-30 所示。

可以看到 HostVG 已经在 Storage Pool 的列表中了，选择 HostVG，如图 1-31 所示。

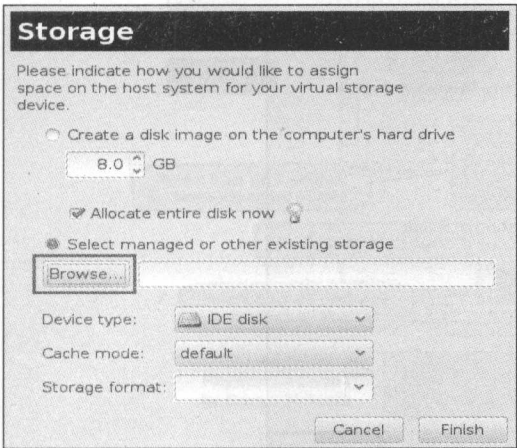


图 1-30

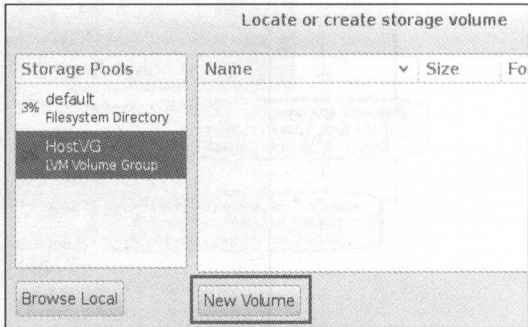


图 1-31

单击“New Volume”，为 volume 命名为 newlv，并设置大小 100MB，如图 1-32 所示。

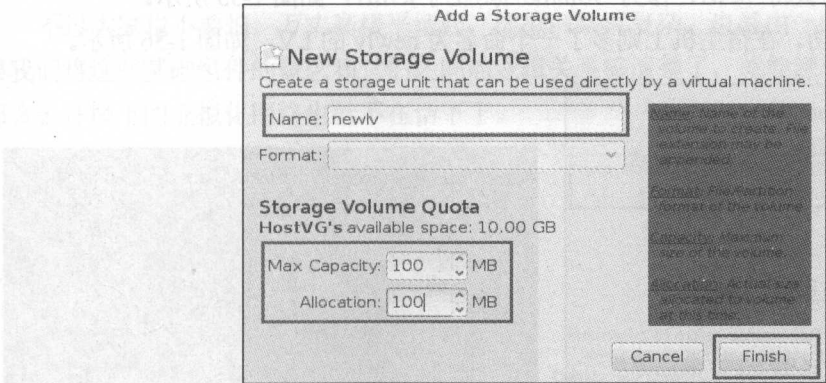


图 1-32



单击“Finish”，newlv 创建成功，如图 1-33 所示。

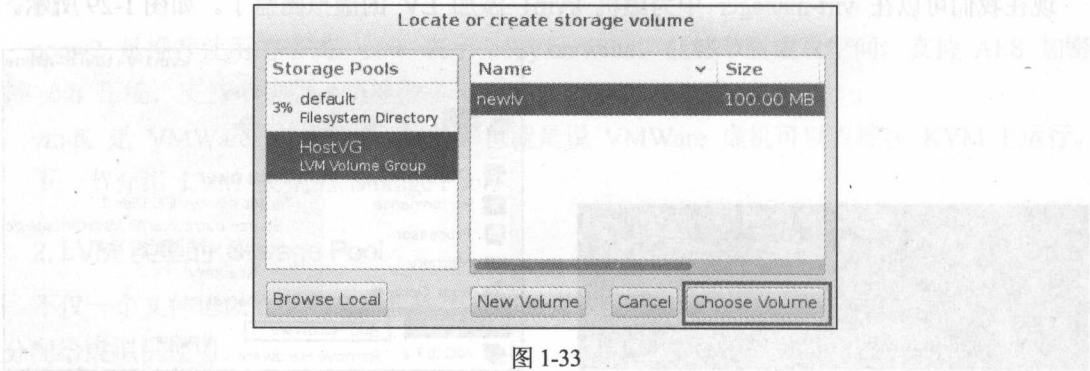


图 1-33

单击“Choose Volume”，如图 1-34 所示。

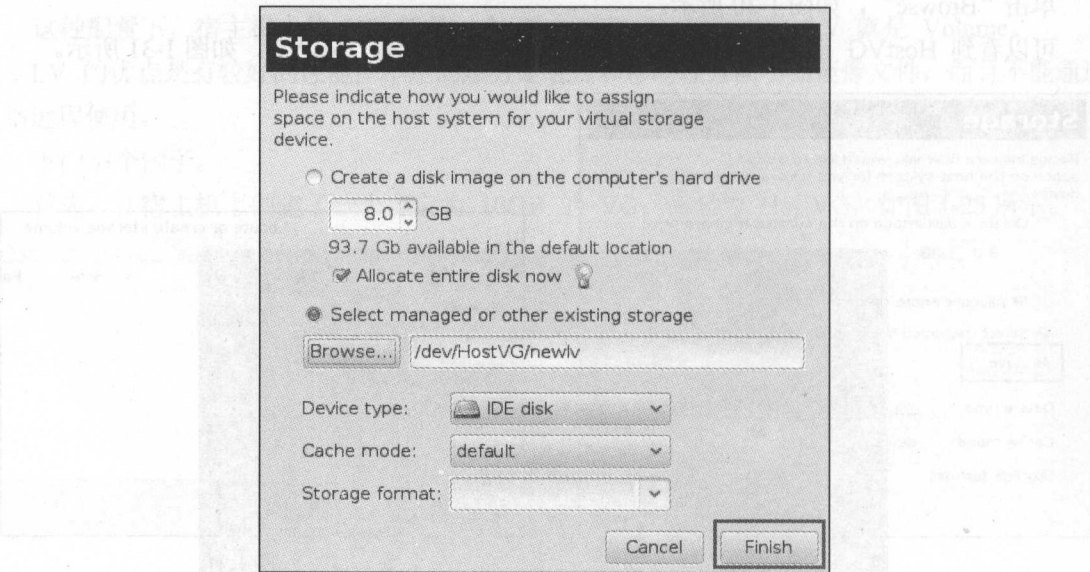


图 1-34

单击“Finish”，确认将 newlv 作为 volume 添加到 kvm1，如图 1-35 所示。

新 volume 添加成功。在宿主机上则多了一个命名为 newlv 的 LV，如图 1-36 所示。

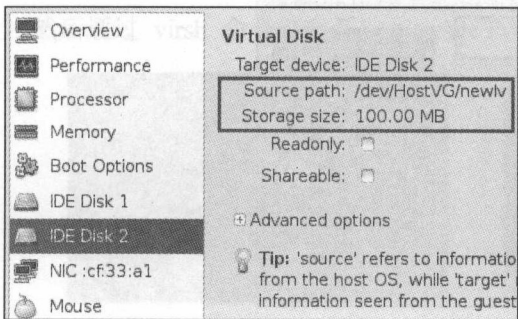


图 1-35

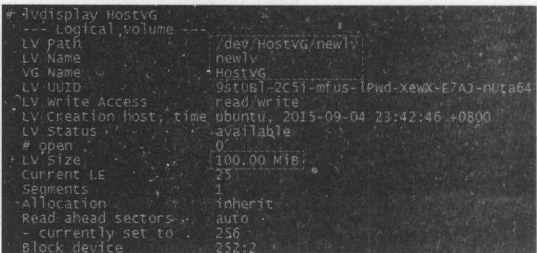


图 1-36

### 3. 其他类型的 Storage Pool

KVM 还支持 iSCSI、Ceph 等多种类型的 Storage Pool，这里就不一一介绍了，最常用的就是目录类型，其他类型可以参考文档 <http://libvirt.org/storage.html>。

下一节我们将开始讨论 KVM 的网络虚拟化原理。

## 1.5 网络虚拟化

网络虚拟化是虚拟化技术中最复杂的部分，学习难度最大。但因为网络是虚拟化中非常重要的资源，所以再硬的骨头也必须要把它啃下来。为了让大家对虚拟化网络的复杂程度有一个直观的认识，请看图 1-37 所示。

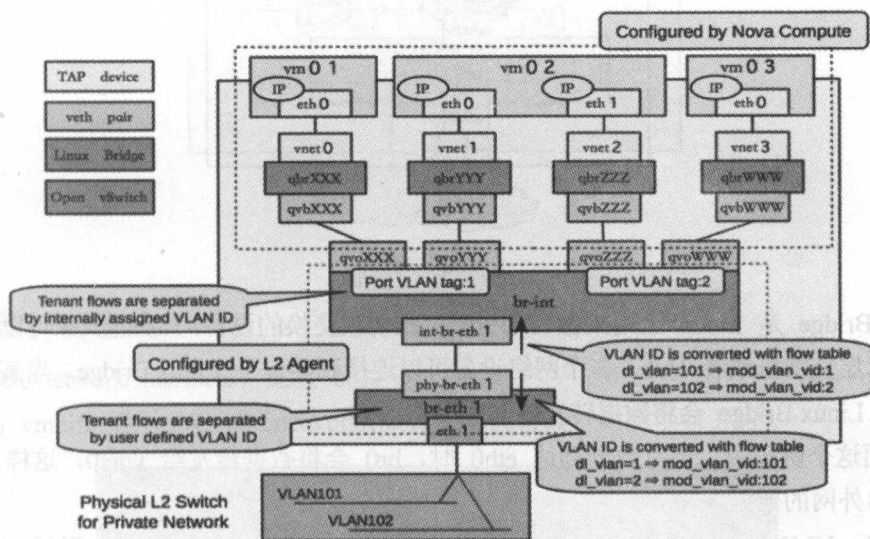


图 1-37

这是 OpenStack 官网上给出的计算节点（可以理解为 KVM 的宿主机）虚拟网络的逻辑图，上面的网络设备很多，层次也很复杂，我第一次看到这张图，也着实被吓了一跳。

不过大家也不要怕，万丈高楼平地起，虚拟网络再复杂，也是由一些基础的组件构成的。只要我们将这些基础组件的概念和它们之间的逻辑关系搞清楚了，就能深刻理解虚拟网络的架构，那么云环境下的虚拟化网络也就不在话下了。

下面我们来学习网络虚拟化中最重要的两个东西：Linux Bridge 和 VLAN。

### 1.5.1 Linux Bridge

#### 1. 基本概念

假设宿主机有 1 块与外网连接的物理网卡 eth0，上面跑了 1 个虚机 VM1，现在有个问题是：

如何让 VM1 能够访问外网？

至少有两种方案：

(1) 将物理网卡 `eth0` 直接分配给 VM1 。

但实施这个方案随之带来的问题很多：宿主机就没有网卡，无法访问了；新的虚机，比如 VM2 也没有网卡。

(2) 给 VM1 分配一个虚拟网卡 `vnet0`，通过 Linux Bridge `br0` 将 `eth0` 和 `vnet0` 连接起来，如图 1-38 所示。这个是我们推荐的方案。

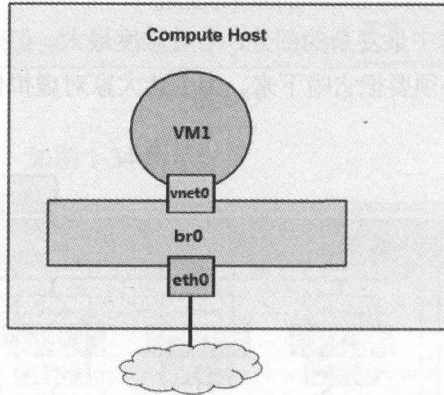


图 1-38

Linux Bridge 是 Linux 上用来做 TCP/IP 二层协议交换的设备，其功能大家可以简单地理解为是一个二层交换机或者 Hub。多个网络设备可以连接到同一个 Linux Bridge，当某个设备收到数据包时，Linux Bridge 会将数据转发给其他设备。

在上面这个例子中，当有数据到达 `eth0` 时，`br0` 会将数据转发给 `vnet0`，这样 VM1 就能接收到来自外网的数据。

反过来，VM1 发送数据给 `vnet0`，`br0` 也会将数据转发到 `eth0`，从而实现了 VM1 与外网的通信。

现在我们增加一个虚机 VM2，如图 1-39 所示。

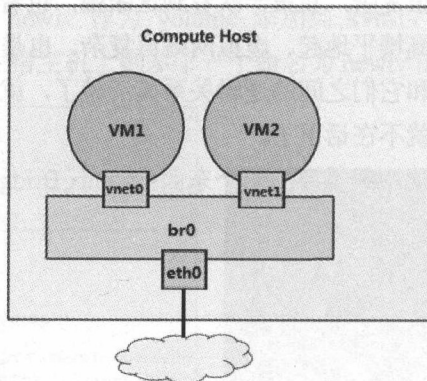


图 1-39

VM2 的虚拟网卡 vnet1 也连接到了 br0 上。现在 VM1 和 VM2 之间可以通信，同时 VM1 和 VM2 也都可以与外网通信。

有了上面的基础支持，下一节将演示如何在实验环境中实现这套虚拟网络。

## 2. 动手实践虚拟网络

本节将演示如何在实验环境中实现如图 1-40 所示的虚拟网络。

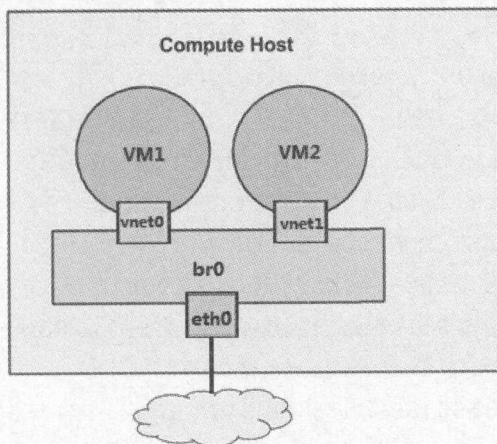


图 1-40

## 3. 配置 Linux Bridge br0

编辑 /etc/network/interfaces，配置 br0。

下面用 vmdiff 展示了对 /etc/network/interfaces 的修改，如图 1-41 所示。

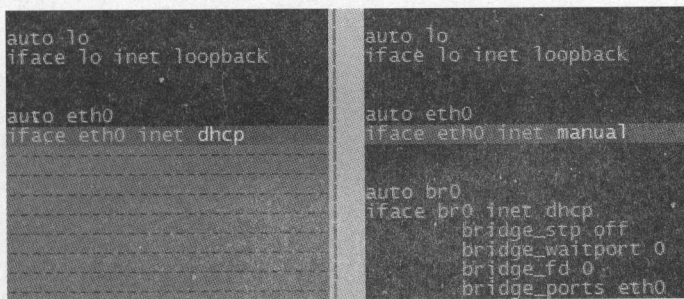


图 1-41

有两点需要注意：

- 之前宿主机的 IP 是通过 dhcp 配置在 eth0 上的；创建 Linux Bridge 之后，IP 就必须放到 br0 上了。
- 在 br0 的配置信息中请注意最后一行 bridge\_ports eth0，其作用就是将 eth0 挂到 br0 上。

重启宿主机，查看 IP 配置，可以看到 IP 已经放到 br0 上了。



```

# ifconfig
br0      Link encap:Ethernet HWaddr 00:0c:29:8d:ec:be
         inet      addr:192.168.111.107          Bcast:192.168.111.255
Mask:255.255.255.0
         inet6 addr: fe80::20c:29ff:fe8d:ecbe/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
         RX packets:22573 errors:0 dropped:0 overruns:0 frame:0
         TX packets:2757 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:0
         RX bytes:4927580 (4.9 MB) TX bytes:368895 (368.8 KB)
eth0     Link encap:Ethernet HWaddr 00:0c:29:8d:ec:be
         inet6 addr: fe80::20c:29ff:fe8d:ecbe/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
         RX packets:24388 errors:0 dropped:0 overruns:0 frame:0
         TX packets:2816 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:5590438 (5.5 MB) TX bytes:411558 (411.5 KB)
lo       Link encap:Local Loopback
         inet addr:127.0.0.1 Mask:255.0.0.0
         inet6 addr: ::1/128 Scope:Host
         UP LOOPBACK RUNNING MTU:65536 Metric:1
         RX packets:146 errors:0 dropped:0 overruns:0 frame:0
         TX packets:146 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:0
         RX bytes:10521 (10.5 KB) TX bytes:10521 (10.5 KB)
virbr0   Link encap:Ethernet HWaddr 72:db:fb:f2:19:91
         inet      addr:192.168.122.1          Bcast:192.168.122.255
Mask:255.255.255.0
         UP BROADCAST MULTICAST MTU:1500 Metric:1
         RX packets:0 errors:0 dropped:0 overruns:0 frame:0
         TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:0
         RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

```

用 `brctl show` 查看当前 Linux Bridge 的配置。eth0 已经挂到 br0 上了。

```

# brctl show

```

bridge name	bridge id	STP enabled	interfaces
br0	8000.000c298decbe	no	eth0
virbr0	8000.000000000000	yes	

除了 br0，大家应该注意到还有一个 virbr0 的 Bridge，而且 virbr0 上已经配置了 IP 地址 192.168.122.1。

virbr0 的作用我们会在后面介绍。

在宿主机中，CloudMan（作者）已经提前创建好了虚机 VM1 和 VM2，现在都处于关机状态。

```
# virsh list --all
```

Id	Name	State
-	VM1	shut off
-	VM2	shut off

#### 4. 配置 VM1

下面我们在 virt-manager 中查看一下 VM1 的网卡配置，如图 1-42 所示。为了使大家能够熟练使用命令行工具 virsh 和图形工具 virt-manager，CloudMan 在演示的时候会同时用到它们，两个工具都很重要。

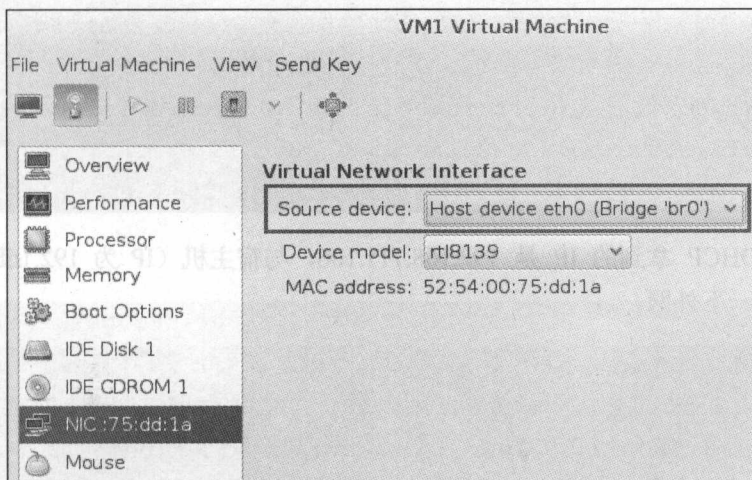


图 1-42

可以看到虚拟网卡的 source device，我们选择的是 br0。

下面我们启动 VM1，看会发生什么？

```
# virsh start VM1
```

```
Domain VM1 started
```

```
# brctl show
```

bridge name	bridge id	STP enabled	interfaces
br0	8000.000c298decbe	no	eth0

```
vnet0
```



```
virbr0      8000.000000000000      yes
```

brctl show 告诉我们, br0 下面添加了一个 vnet0 设备, 通过 virsh 确认这就是 VM1 的虚拟网卡。

```
# virsh domiflist VM1
Interface Type      Source      Model      MAC
-----
vnet0      bridge      br0         rtl8139     52:54:00:75:dd:1a
```

VM1 的 IP 是 DHCP 获得的 (设置静态 IP 当然也可以), 通过 virt-manager 控制台登录 VM1, 查看 IP。

```
# ifconfig
eth0      Link encap:Ethernet HWaddr 52:54:00:75:dd:1a
          inet      addr:192.168.111.106      Bcast:192.168.111.255
Mask:255.255.255.0
          inet6 addr: fe80::5054:ff:fe75:dd1a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:400 errors:0 dropped:0 overruns:0 frame:0
          TX packets:101 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:41950 (41.9 KB) TX bytes:12583 (12.5 KB)
```

VM1 通过 DHCP 拿到的 IP 是 192.168.111.106, 与宿主机 (IP 为 192.168.111.107) 是同一个网段。Ping 一下外网。

```
root@VM1:~# ping www.baidu.com
PING www.a.shifen.com (180.97.33.108) 56(84) bytes of data.
64 bytes from 180.97.33.108: icmpseq=1 ttl=53 time=34.9 ms
64 bytes from 180.97.33.108: icmpseq=2 ttl=53 time=36.2 ms
64 bytes from 180.97.33.108: icmpseq=3 ttl=53 time=38.8 ms
```

没问题, 可以访问。

另外, 在 VM1 中虚拟网卡是 eth0, 并不是 vnet0。

vnet0 是该虚拟网卡在宿主机中对应的设备名称, 其类型是 TAP 设备, 这里需要注意一下。

## 5. 配置 VM2

跟 VM1 一样, VM2 的虚拟网卡也挂在 br0 上, 启动 VM2, 查看网卡信息。

```
# virsh start VM2
Domain VM2 started
```

```
# brctl show
bridge name      bridge id                STP enabled  interfaces
br0              8000.000c298decbe       no          eth0
               vnet0
               vnet1

virbr0          8000.0000000000000      yes
```

br0 下面多了 vnet1, 通过 virsh 确认这就是 VM2 的虚拟网卡。

```
# virsh domiflist VM2
Interface Type      Source      Model      MAC
-----
vnet1     bridge    br0         rtl8139    52:54:00:cf:33:a1
```

VM2 通过 DHCP 拿到的 IP 是 192.168.111.108, 登录 VM2, 验证网络的连通性  
Ping VM1:

```
root@VM2:~# ping VM1
PING VM1 (192.168.111.106) 56(84) bytes of data.
64 bytes from 192.168.111.106: icmpseq=1 ttl=64 time=4.54 ms
64 bytes from 192.168.111.106: icmpseq=2 ttl=64 time=1.63 ms
64 bytes from 192.168.111.106: icmpseq=3 ttl=64 time=2.16 ms
```

Ping 宿主机:

```
root@VM2:~# ping 192.168.111.107
PING 192.168.111.107 (192.168.111.107) 56(84) bytes of data.
64 bytes from 192.168.111.107: icmpseq=1 ttl=64 time=1.02 ms
64 bytes from 192.168.111.107: icmpseq=2 ttl=64 time=0.052 ms
64 bytes from 192.168.111.107: icmpseq=3 ttl=64 time=0.064 ms
```

Ping 外网:

```
root@VM2:~# ping www.baidu.com
PING www.a.shifen.com (180.97.33.107) 56(84) bytes of data.
64 bytes from 180.97.33.107: icmpseq=1 ttl=53 time=53.9 ms
64 bytes from 180.97.33.107: icmpseq=2 ttl=53 time=45.0 ms
64 bytes from 180.97.33.107: icmpseq=3 ttl=53 time=44.2 ms
```

可见, 通过 br0 这个 Linux Bridge, 我们实现了 VM1、VM2、宿主机和外网这四者之间的数据通信。

## 6. 理解 virbr0

virbr0 是 KVM 默认创建的一个 Bridge，其作用是为连接其上的虚拟网卡提供 NAT 访问外网的功能。

virbr0 默认分配了一个 IP 192.168.122.1，并为连接其上的其他虚拟网卡提供 DHCP 服务。

下面我们演示如何使用 virbr0。

在 virt-manager 打开 VM1 的配置界面，网卡 Source device 选择 “default”，将 VM1 的网卡挂在 virbr0 上，如图 1-43 所示。

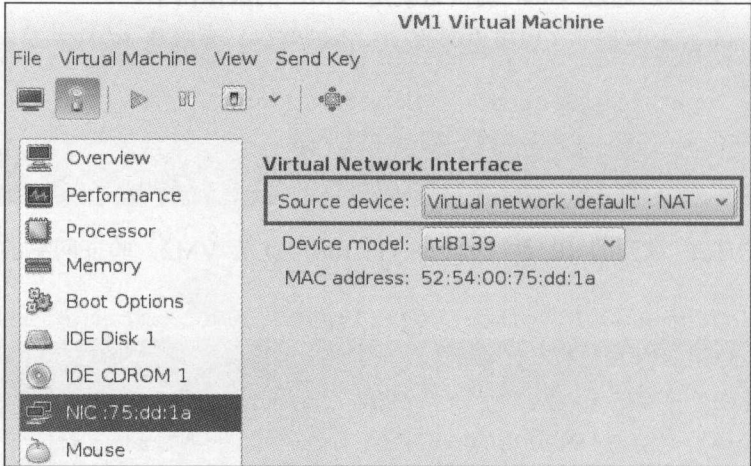


图 1-43

启动 VM1，brctl show 可以查看到 vnet0 已经挂在了 virbr0 上。

```
# brctl show
bridge name      bridge id                STP enabled    interfaces
br0               8000.000c298decbe        no             eth0
virbr0            8000.fe540075dd1a        yes            vnet0
```

用 virsh 命令确认 vnet 就是 VM1 的虚拟网卡。

```
# virsh domiflist VM1
Interface Type      Source      Model      MAC
-----
vnet0      network    default     rtl8139     52:54:00:75:dd:1a
```

virbr0 使用 dnsmasq 提供 DHCP 服务，可以在宿主机中查看该进程信息。

```
# ps -elf|grep dnsmasq
5 S libvirt+ 2422    1 0 80    0 - 7054 poll_s 11:26 ?      00:00:00
/usr/sbin/dnsmasq --conf-file=/var/lib/libvirt/dnsmasq/default.conf
```



在 `/var/lib/libvirt/dnsmasq/` 目录下有一个 `default.leases` 文件, 当 VM1 成功获得 DHCP 的 IP 后, 可以在该文件中查看到相应的信息。

```
# cat /var/lib/libvirt/dnsmasq/default.leases
1441525677 52:54:00:75:dd:1a 192.168.122.6 ubuntu *
```

上面显示 192.168.122.6 已经分配给 MAC 地址为 **52:54:00:75:dd:1a** 的网卡, 这正是 `vnet0` 的 MAC。之后就可以使用该 IP 访问 VM1 了。

```
# ssh 192.168.122.6
root@192.168.122.6's password:
Welcome to Ubuntu 14.04.2 LTS (GNU/Linux 3.16.0-30-generic x86_64)
Last login: Sun Sep 6 01:30:23 2015
root@VM1:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 52:54:00:75:dd:1a
          inet      addr:192.168.122.6          Bcast:192.168.122.255
Mask:255.255.255.0
          inet6 addr: fe80::5054:ff:fe75:dd1a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:61 errors:0 dropped:0 overruns:0 frame:0
          TX packets:66 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:7453 (7.4 KB)  TX bytes:8649 (8.6 KB)
```

Ping 一下外网。

```
root@VM1:~# ping www.baidu.com
PING www.a.shifen.com (180.97.33.107) 56(84) bytes of data.
64 bytes from 180.97.33.107: icmpseq=1 ttl=52 time=36.9 ms
64 bytes from 180.97.33.107: icmpseq=2 ttl=52 time=119 ms
64 bytes from 180.97.33.107: icmpseq=3 ttl=52 time=88.5 ms
64 bytes from 180.97.33.107: icmpseq=4 ttl=52 time=38.0 ms
64 bytes from 180.97.33.107: icmpseq=5 ttl=52 time=122 ms
```

没有问题, 可以访问外网, 说明 NAT 起作用了。

需要说明的是, 使用 NAT 的虚机 VM1 可以访问外网, 但外网无法直接访问 VM1。

因为 VM1 发出的网络包源地址并不是 192.168.122.6, 而是被 NAT 替换为宿主机的 IP 地址了。

这个与使用 `br0` 不一样, 在 `br0` 的情况下, VM1 通过自己的 IP 直接与外网通信, 不会经过 NAT 地址转换。

## 1.5.2 VLAN

### 1. 基本概念

LAN 表示 Local Area Network，本地局域网，通常使用 Hub 和 Switch 来连接 LAN 中的计算机。

一般来说，两台计算机连入同一个 Hub 或者 Switch 时，它们就在同一个 LAN 中。

一个 LAN 表示一个广播域，其含义是：LAN 中的所有成员都会收到任意一个成员发出的广播包。

VLAN 表示 Virtual LAN。一个带有 VLAN 功能的 switch 能够将自己的端口划分出多个 LAN。

计算机发出的广播包可以被同一个 LAN 中其他计算机收到，但位于其他 LAN 的计算机则无法收到。

简单地讲，VLAN 将一个交换机分成了多个交换机，限制了广播的范围，在二层上将计算机隔离到不同的 VLAN 中。

比方说，有两组机器，Group A 和 B。我们想配置成 Group A 中的机器可以相互访问，Group B 中的机器也可以相互访问，但是 A 和 B 中的机器无法互相访问。

一种方法是使用两个交换机，A 和 B 分别接到一个交换机。

另一种方法是使用一个带 VLAN 功能的交换机，将 A 和 B 的机器分别放到不同的 VLAN 中。

请注意，VLAN 的隔离是二层上的隔离，A 和 B 无法相互访问指的是二层广播包（比如 arp）无法跨越 VLAN 的边界。

但在三层上（比如 IP）是可以通过路由器让 A 和 B 互通的。概念上一定要分清。

现在的交换机几乎都是支持 VLAN 的。

通常交换机的端口有两种配置模式：Access 和 Trunk，如图 1-44 所示。

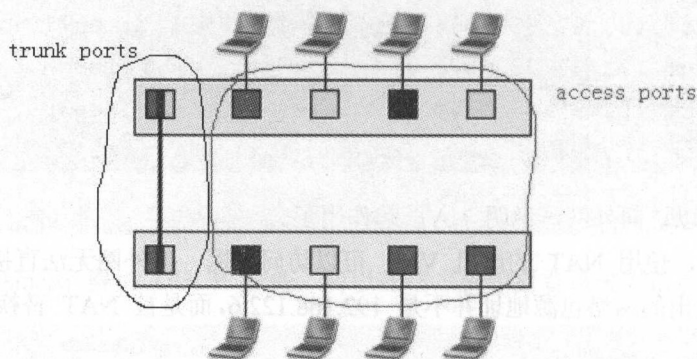


图 1-44

### ● Access 口

这些端口被打上了 VLAN 的标签，表明该端口属于哪个 VLAN。

不同 VLAN 用 VLAN ID 来区分，VLAN ID 的范围是 1~4096。

Access 口都是直接与计算机网卡相连的，这样从该网卡出来的数据包流入 Access 口后，就会被打上了所在 VLAN 的标签。

Access 口只能属于一个 VLAN。

### ● Trunk 口

假设有两个交换机 A 和 B。

A 上有 VLAN1 (红)、VLAN2 (黄)、VLAN3 (蓝)；B 上也有 VLAN1、VLAN 2、VLAN 3。

那如何让 A 和 B 上相同 VLAN 之间能够通信呢？办法是将 A 和 B 连起来，而且连接 A 和 B 的端口要允许 VLAN1、2、3 三个 VLAN 的数据都能够通过。这样的端口就是 Trunk 口了。

VLAN1、2、3 的数据包在通过 Trunk 口到达对方交换机的过程中始终带着自己的 VLAN 标签。

了解了 VLAN 的概念之后，我们来看 KVM 虚拟化环境下是如何实现 VLAN 的。还是先看图，如图 1-45 所示。

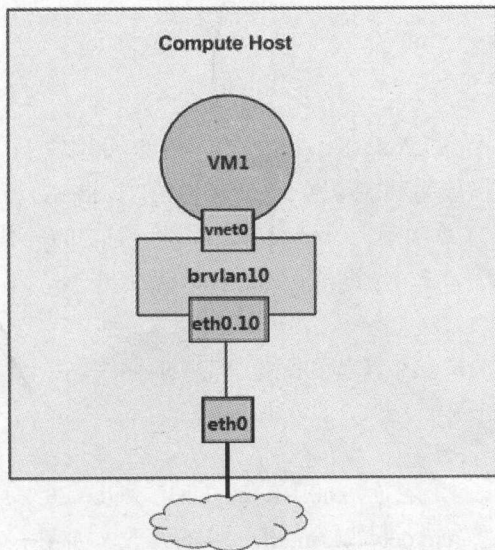


图 1-45

eth0 是宿主机上的物理网卡，有一个命名为 eth0.10 的子设备与之相连。

eth0.10 就是 VLAN 设备了，其 VLAN ID 就是 VLAN 10。

eth0.10 挂在命名为 brvlan10 的 Linux Bridge 上，虚拟机 VM1 的虚拟网卡 vent0 也挂在 brvlan10 上。

这样的配置，其效果就是：

- 宿主机用软件实现了一个交换机（当然是虚拟的），上面定义了一个 VLAN10。
- eth0.10, brvlan10 和 vnet0 都分别接到 VLAN10 的 Access 口上。而 eth0 就是一个 Trunk 口。
- VM1 通过 vnet0 发出来的数据包会被打上 VLAN10 的标签。

eth0.10 的作用是：定义了 VLAN10。

brvlan10 的作用是：Bridge 上的其他网络设备自动加入到 VLAN10 中。

我们再增加一个 VLAN20，如图 1-46 所示。

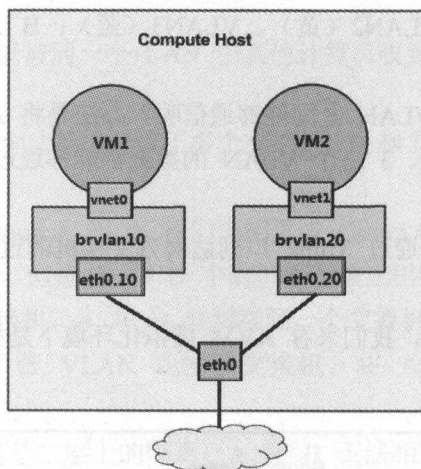


图 1-46

这样虚拟交换机就有两个 VLAN 了，VM1 和 VM2 分别属于 VLAN10 和 VLAN20。

对于新创建的虚机，只需要将其虚拟网卡放入相应的 Bridge，就能控制其所属的 VLAN。

VLAN 设备总是以母子关系出现，母子设备之间是一对多的关系。

一个母设备（eth0）可以有多个子设备（eth0.10, eth0.20 .....），而一个子设备只有一个母设备。

下面我们来看如何在实验环境中实施和配置 VLAN 网络。

## 2. 配置 VLAN

编辑 /etc/network/interfaces，配置 eth0.10、brvlan10、eth0.20 和 brvlan20。

下面用 vmdiff 展示了对 /etc/network/interfaces 的修改，如图 1-47 所示。



```

auto lo
iface lo inet loopback

auto eth0
iface eth0 inet dhcp

auto eth0.10
iface eth0.10 inet manual
vlan-raw-device eth0

auto brvlan10
iface brvlan10 inet manual
bridge_stp off
bridge_waitport 0
bridge_fd 0
bridge_ports eth0.10

auto eth0.20
iface eth0.20 inet manual
vlan-raw-device eth0

auto brvlan20
iface brvlan20 inet manual
bridge_stp off
bridge_waitport 0
bridge_fd 0
bridge_ports eth0.20

```

图 1-47

重启宿主机，ifconfig 各个网络接口，如图 1-48 所示。

```

# ifconfig
brvlan10 Link encap:Ethernet Hwaddr 00:0c:29:8d:ec:be
inet6 addr: fe80::20c:29ff:fe8d:ecbe/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:37 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:0 (0.0 B) TX bytes:5030 (5.0 KB)

brvlan20 Link encap:Ethernet Hwaddr 00:0c:29:8d:ec:be
inet6 addr: fe80::20c:29ff:fe8d:ecbe/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:37 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:0 (0.0 B) TX bytes:5030 (5.0 KB)

eth0 Link encap:Ethernet Hwaddr 00:0c:29:8d:ec:be
inet addr:192.168.236.130 Bcast:192.168.236.255 Mask:
inet6 addr: fe80::20c:29ff:fe8d:ecbe/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:1991 errors:0 dropped:0 overruns:0 frame:0
TX packets:2365 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:210583 (210.5 KB) TX bytes:1989780 (1.9 MB)

eth0.10 Link encap:Ethernet Hwaddr 00:0c:29:8d:ec:be
inet6 addr: fe80::20c:29ff:fe8d:ecbe/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:74 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:0 (0.0 B) TX bytes:10060 (10.0 KB)

eth0.20 Link encap:Ethernet Hwaddr 00:0c:29:8d:ec:be
inet6 addr: fe80::20c:29ff:fe8d:ecbe/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:74 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:0 (0.0 B) TX bytes:10060 (10.0 KB)

```

图 1-48

用 `brctl show` 查看当前 Linux Bridge 的配置。

`eth0.10` 和 `eth0.20` 分别挂在 `brvlan10` 和 `brvlan20` 上了，如图 1-49 所示。

```
# brctl show
bridge name      bridge id        STP enabled    interfaces
brvlan10         8000.000c298decbe  no            eth0.10
brvlan20         8000.000c298decbe  no            eth0.20
virbr0           8000.000000000000  yes
```

图 1-49

在宿主机中已经提前创建好了虚机 VM1 和 VM2，现在都处于关机状态，如图 1-50 所示。

```
# virsh list --all
Id      Name      State
-----
-       VM1       shut off
-       VM2       shut off
```

图 1-50

3. 配置 VM1

在 virt-manager 中，将 VM1 的虚拟网卡挂到 brvlan10 上，如图 1-51 所示。

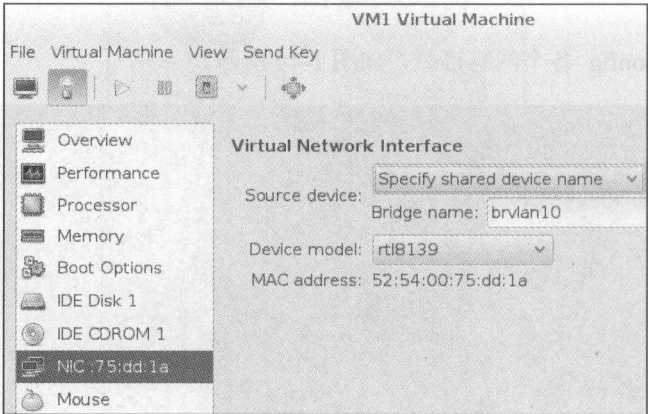


图 1-51

启动 VM1，如图 1-52 所示。

```
# virsh start VM1
Domain VM1 started
```

图 1-52

查看 Bridge，发现 brvlan10 已经连接了一个 vnet0 设备，如图 1-53 所示。

```
# brctl show
bridge name      bridge id        STP enabled    interfaces
brvlan10         8000.000c298decbe  no            vnet0
brvlan20         8000.000c298decbe  no            eth0.20
virbr0           8000.000000000000  yes
```

图 1-53

通过 virsh 确认这就是 VM1 的虚拟网卡，如图 1-54 所示。

```
# virsh domiflist VM1
Interface Type      Source      Model      MAC
-----
vnet0      bridge      brvlan10    rtl8139     52:54:00:75:dd:1a
```

图 1-54

#### 4. 配置 VM2

类似的，将 VM2 的网卡挂在 brvlan20 上，如图 1-55 所示。

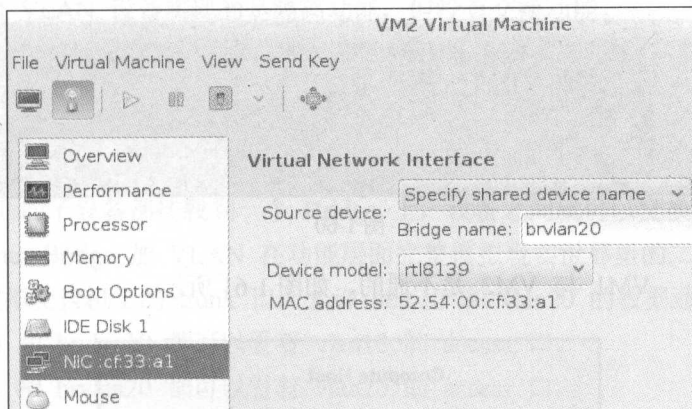


图 1-55

启动 VM2，如图 1-56 所示。

```
# virsh start VM2
Domain VM2 started
```

图 1-56

查看 Bridge，发现 brvlan20 已经连接了一个 vnet1 设备，如图 1-57 所示。

```
# brctl show
bridge name      bridge id        STP enabled      interfaces
brvlan10         8000.000c298decbe  no               vnet0 eth0.10
brvlan20         8000.000c298decbe  no               vnet1 eth0.20
virbr0           8000.000000000000  yes
```

图 1-57

通过 virsh 确认这就是 VM2 的虚拟网卡，如图 1-58 所示。

```
# virsh domiflist VM2
Interface Type      Source      Model      MAC
-----
vnet1      bridge      brvlan20    rtl8139     52:54:00:cf:33:a1
```

图 1-58

#### 5. 验证 VLAN 的隔离性

为了验证 VLAN10 和 VLAN20 之间的隔离，我们为 VM1 和 VM2 配置同一网段的 IP。配置 VM1 的 IP，如图 1-59 所示。

```
root@VM1:~# ifconfig eth0 192.168.100.10 netmask 255.255.255.0 up
root@VM1:~# ifconfig
eth0      *Link encap:Ethernet  HWaddr 52:54:00:75:dd:1a
          inet addr:192.168.100.10  Bcast:192.168.111.255  Mask:255.255.255.0
          inet6 addr: fe80::3054:ff:fe75:dd1a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:400 errors:0 dropped:0 overruns:0 frame:0
          TX packets:101 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:41950 (41.9 KB)  TX bytes:12583 (12.5 KB)
```

图 1-59

配置 VM2 的 IP，如图 1-60 所示。

```
root@VM2:~# ifconfig eth0 192.168.100.20 netmask 255.255.255.0 up
root@VM2:~# ifconfig
eth0      *Link encap:Ethernet  HWaddr 52:54:00:cf:33:a1
          inet addr:192.168.100.20  Bcast:192.168.111.255  Mask:255.255.255.0
          inet6 addr: fe80::3054:ff:fe75:dd1a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:300 errors:0 dropped:0 overruns:0 frame:0
          TX packets:89 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:27950 (27.9 KB)  TX bytes:12583 (8.5 KB)
```

图 1-60

Ping 测试结果： VM1 与 VM2 是不通的，如图 1-61 所示。

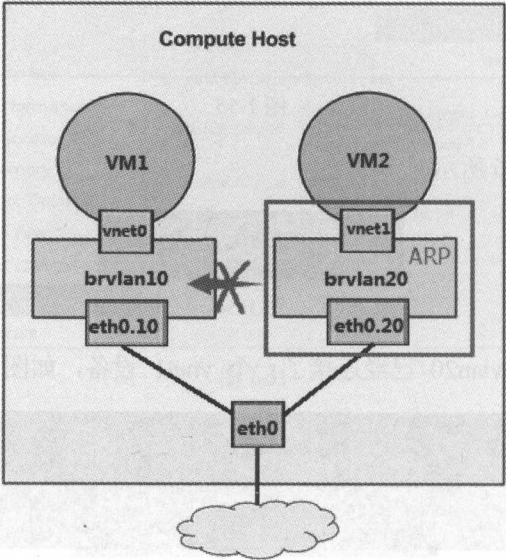


图 1-61

原因如下：

- (1) VM2 向 VM1 发 Ping 包之前，需要知道 VM1 的 IP 192.168.100.10 所对应的 MAC 地址。VM2 会在网络上广播 ARP 包，其作用就是询问“谁知道 192.168.100.10 的 MAC 地址是多少？”
- (2) ARP 是二层协议，VLAN 的隔离作用使得 ARP 只能在 VLAN20 范围内广播，只有 brvlan20 和 eth0.20 能收到，VLAN10 里的设备是收不到的。VM1 无法应答 VM2 发出的 ARP 包。
- (3) VM2 拿不到 VM1 vnet0 的 MAC 地址，也就 Ping 不到 VM1。



### 1.5.3 Linux Bridge + VLAN = 虚拟交换机

现在对 KVM 的网络虚拟化做个总结。

(1) 物理交换机存在多个 VLAN，每个 VLAN 拥有多个端口。

同一 VLAN 端口之间可以交换转发，不同 VLAN 端口之间隔离。所以交换机包含两层功能：交换与隔离。

(2) Linux 的 VLAN 设备实现的是隔离功能，但没有交换功能。

一个 VLAN 母设备（比如 eth0）不能拥有两个相同 ID 的 VLAN 子设备，因此也就不可能出现数据交换情况。

(3) Linux Bridge 专门实现交换功能。

将同一 VLAN 的子设备都挂载到一个 Bridge 上，设备之间就可以交换数据了。

总结起来，Linux Bridge 加 VLAN 在功能层面完整模拟现实世界里的二层交换机。

eth0 相当于虚拟交换机上的 trunk 口，允许 vlan10 和 vlan20 的数据通过。

eth0.10, vnet0 和 brvlan10 都可以看着 vlan10 的 access 口。

eth0.20, vnet1 和 brvlan20 都可以看着 vlan20 的 access 口。

# 第 2 章

## ◀ 云 计 算 ▶

“云计算”算是近年来最热的词了。现在 IT 行业见面不说这三个字您都不好意思跟人家打招呼。对于云计算，学术界有各种定义，大家有兴趣的可以百度一下。CloudMan 在这里主要想从技术的角度谈谈对云计算的理解。

### 2.1 基本概念

所有的新事物都不是突然冒出来的，都有前世和今生。云计算也是 IT 技术不断发展的产物。要理解云计算，需要对 IT 系统架构的发展过程有所认识。

请看图 2-1 所示。

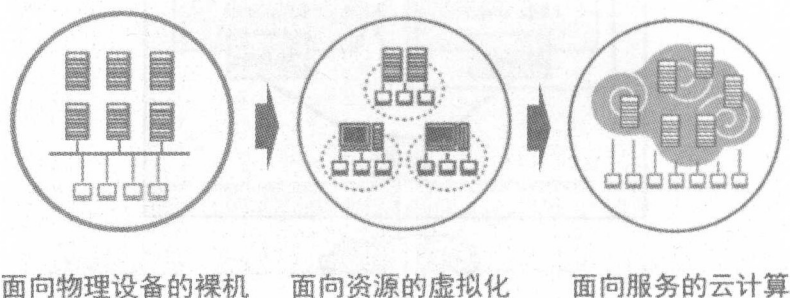


图 2-1

IT 系统架构的发展到目前为止大致可以分为 3 个阶段：

#### 1. 物理机架构

这一阶段，应用部署和运行在物理机上。比如企业要上一个 ERP 系统，如果规模不大，可以找 3 台物理机，分别部署 Web 服务器、应用服务器和数据库服务器。

如果规模大一点，各种服务器可以采用集群架构，但每个集群成员也还是直接部署在物理机上。

我见过的客户早期都是这种架构，一套应用一套服务器，通常系统的资源使用率都很低，达到 20% 的都是好的。

## 2. 虚拟化架构

摩尔定律决定了物理服务器的计算能力越来越强，虚拟化技术的发展大大提高了物理服务器的资源使用率。

这个阶段，物理机上运行若干虚拟机，应用系统直接部署到虚拟机上。

虚拟化的好处还体现在减少了需要管理的物理机数量，同时节省了维护成本。

## 3. 云计算架构

虚拟化提高了单台物理机的资源使用率，随着虚拟化技术的应用，IT 环境中越来越多的虚拟机，这时新的需求产生了：

如何对 IT 环境中的虚拟机进行统一和高效的管理。

有需求就有供给，云计算登上了历史舞台。

计算（CPU/内存）、存储和网络是 IT 系统的三类资源。通过云计算平台，这三类资源变成了三个池子。当需要虚机的时候，只需要向平台提供虚机的规格。平台会快速地从三个资源池分配相应的资源，部署出这样一个满足规格的虚机。虚机的使用者不再需要关心虚机运行在哪里，存储空间从哪里来，IP 是如何分配，这些云平台都搞定了。

云平台是一个面向服务的架构，按照提供服务的不同分为 IaaS、PaaS 和 SaaS。

请看图 2-2 所示。

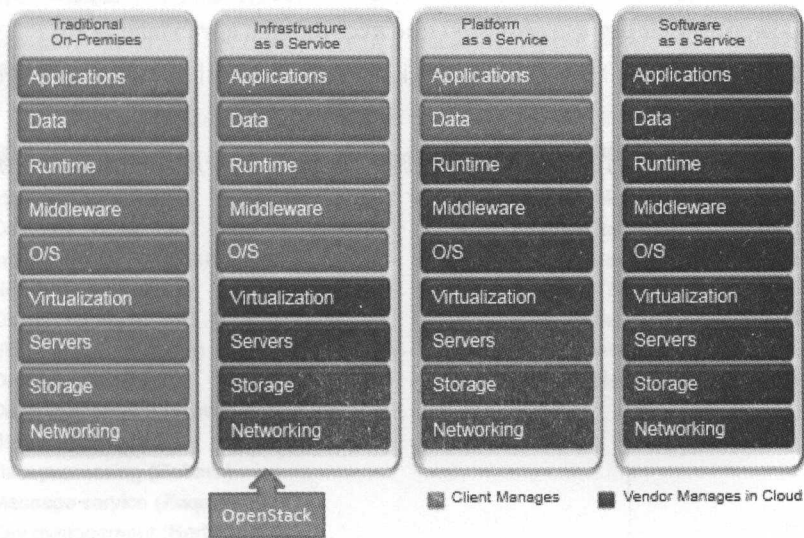


图 2-2

- IaaS (Infrastructure as a Service) 提供的服务是虚拟机。

IaaS 负责管理虚机的生命周期，包括创建、修改、备份、启停、销毁等。

使用者从云平台得到的是一个已经安装好镜像（操作系统+其他预装软件）的虚拟机。使用者需要关心虚机的类型（OS）和配置（CPU、内存、磁盘），并且自己负责部署上层的中间件和应用。

IaaS 的使用者通常是数据中心的系统管理员。典型的 IaaS 例子有 AWS、Rackspace、阿里云等。

- PaaS (Platform as a Service) 提供的服务是应用的运行环境和一系列中间件服务 (比如数据库、消息队列等)。

使用者只需专注应用的开发, 并将自己的应用和数据部署到 PaaS 环境中。

PaaS 负责保证这些服务的可用性和性能。PaaS 的使用者通常是应用的开发人员。典型的 PaaS 有 Google App Engine、IBM BlueMix 等。

- SaaS (Software as a Service) 提供的是应用服务。

使用者只需要登录并使用应用, 无须关心应用使用什么技术实现, 也不需要关心应用部署在哪里。SaaS 的使用者通常是应用的最终用户。典型的 SaaS 有 Google Gmail、Salesforce 等。

## 2.2 云计算和 OpenStack

OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed through a dashboard that gives administrators control while empowering their users to provision resources through a web interface.

以上是官网对 OpenStack 的定义, OpenStack 对数据中心的计算、存储和网络资源进行统一管理。

由此可见, OpenStack 针对的是 IT 基础设施, 是 IaaS 这个层次的云操作系统。



# 第二篇

## OpenStack 核心

终于正式进入 OpenStack 部分了。今天开始，CloudMan 将带着大家一步一步揭开 OpenStack 的神秘面纱。

OpenStack 已经走过了 6 个年头。每半年会发布一个版本，版本以字母顺序命名。现在已经到第 12 个版本 Liberty（字母 L）。

OpenStack 最初只有两个模块（服务），现在已经有 20+个模块（如下图所示），每个模块作为独立的子项目开发。

### Main services

- Identity (**Keystone**)
- Compute (**Nova**)
- Image service (**Glance**)
- Networking (**Neutron**)
- Object Storage (**Swift**)
- Block Storage (**Cinder**)
- Orchestration (**Heat**)
- Database Service (**Trove**)
- Bare Metal (**Ironic**)
- Data processing (**Sahara**)
- Message service (**Zaqar**)
- Key management (**Barbican**)
- DNS (**Designate**)
- Shared Filesystems (**Manila**)
- Containers service (**Magnum**)
- Application catalog (**Murano**)
- Governance service (**Congress**)
- Workflow service (**Mistral**)
- Key-value store as a Service (**Magnetodb**)

### Supporting services

- Dashboard (**Horizon**)
- Telemetry (**Ceilometer**)
- Common Libraries (**Oslo**)
- Deployment (**TripleO**)
- Command-line client (**OpenStackClient**)
- Benchmark service (**Rally**)
- Puppet modules (**PuppetOpenStack**)

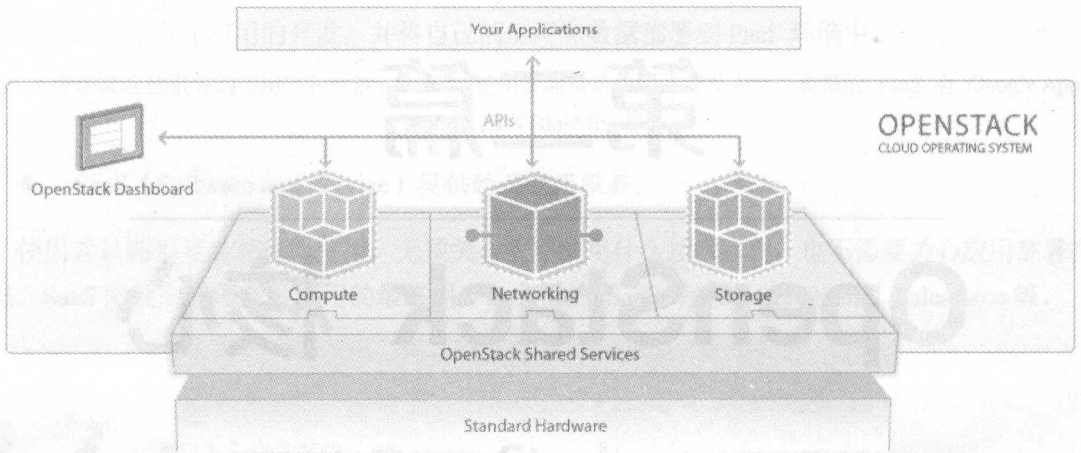
面对如此庞大的阵容，作为初学者我们如何起步呢？

这也是 CloudMan 写这个系列教程的初衷：通过实际操作帮助初学者由浅入深地学习和实践

OpenStack，并最终具备实施 OpenStack 的能力。

我们会把学习的重点放在 OpenStack 最核心的地方。

那什么是核心呢？请看下图。



作为 IaaS 层的云操作系统，OpenStack 为虚拟机提供并管理三大类资源：计算、网络和存储。

这三个就是核心，所以我们的学习重点就是：搞清楚 OpenStack 是如何对计算、网络和存储资源进行管理的。

在 20+ 个模块中，管理这三类资源的核心模块其实不多，这几个模块就是我们的重点了。要达到这个目的，我们自然需要研究 OpenStack 的整体架构。

弄清楚架构里哪些核心模块负责管理计算资源、网络资源和存储资源？模块之间如何协调工作？

同时我们会构建一个实验环境，进到各个模块的内部，通过实际操作真正理解和掌握 OpenStack。

好，下面我们就从架构开始吧。

# 第 3 章

## ◀ OpenStack 架构 ▶

架构是个好东西，它能帮助我们站在高处看清楚事物的整体结构，避免过早地进入细节而迷失方向。

### 3.1 Conceptual Architecture

图 3-1 所示的是 OpenStack 的 Conceptual Architecture（概念架构）。

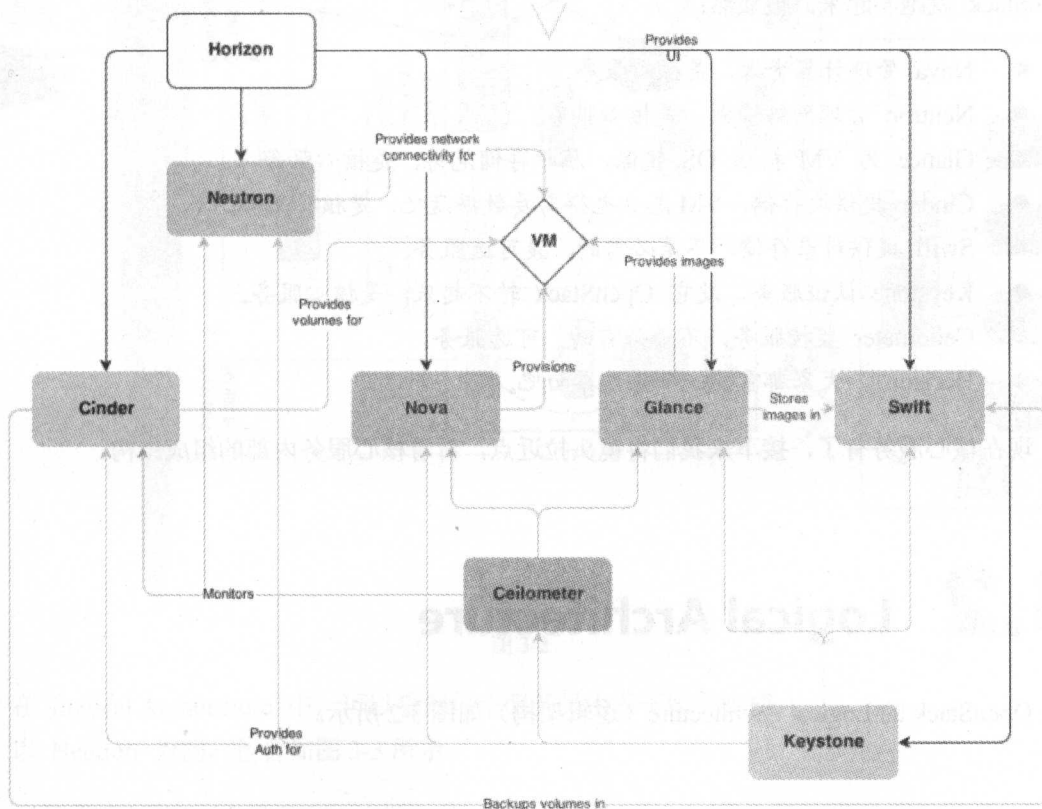


图 3-1



中间菱形是虚拟机，围绕 VM 的那些长方形代表 OpenStack 不同的模块（OpenStack 叫服务，后面都用服务这个术语），下面来分别介绍。

**Nova**：管理 VM 的生命周期，是 OpenStack 中最核心的服务。

**Neutron**：为 OpenStack 提供网络连接服务，负责创建和管理 L2、L3 网络，为 VM 提供虚拟网络和物理网络连接。

**Glance**：管理 VM 的启动镜像，Nova 创建 VM 时将使用 Glance 提供的镜像。

**Cinder**：为 VM 提供块存储服务。Cinder 提供的每一个 Volume 在 VM 看来就是一块虚拟硬盘，一般用作数据盘。

**Swift**：提供对象存储服务。VM 可以通过 RESTful API 存放对象数据。作为可选的方案，Glance 可以将镜像存放在 Swift 中；Cinder 也可以将 Volume 备份到 Swift 中。

**Keystone**：为 OpenStack 的各种服务提供认证和权限管理服务。简单地说，OpenStack 上的每一个操作都必须通过 Keystone 的审核。

**Ceilometer**：提供 OpenStack 监控和计量服务，为报警、统计或计费提供数据。

**Horizon**：为 OpenStack 用户提供一个 Web 的自服务 Portal。

在上面的这些服务中，哪些是 OpenStack 的核心服务呢？核心服务就是如果没有它，OpenStack 就跑不起来了。很显然：

- Nova 管理计算资源，是核心服务。
- Neutron 管理网络资源，是核心服务。
- Glance 为 VM 提供 OS 镜像，属于存储范畴，是核心服务。
- Cinder 提供块存储，VM 怎么也得需要数据盘吧，是核心服务。
- Swift 提供对象存储，不是必需的，是可选服务。
- Keystone 认证服务，没它 OpenStack 转不起来，是核心服务。
- Ceilometer 监控服务，不是必需的，可选服务。
- Horizon，大家都需要一个操作界面吧。

现在核心服务有了，接下来我们将镜头拉近点，看看核心服务内部的组成结构。

## 3.2 Logical Architecture

OpenStack 的 Logical Architecture（逻辑架构）如图 3-2 所示。





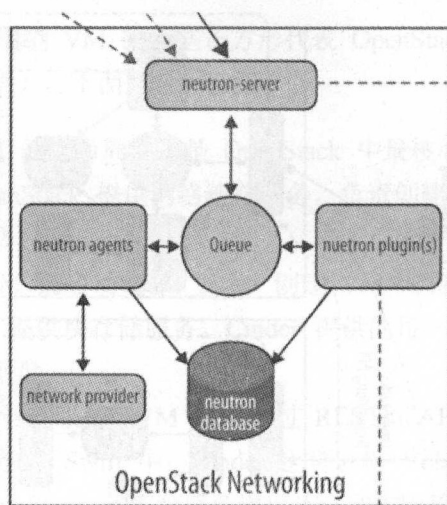


图 3-3

- Neutron Server、Neutron plugins 和 Neutron agents
- Network provider
- 消息队列 Queue
- 数据库 Neutron Database

在后面 Neutron 章节我们会展开学习这些组件。

这里想要强调一点：

上面是 Logical Architecture 描述的是 Neutron 服务各个组成部分以及各组件之间的逻辑关系。

而在实际的部署方案上，各个组件可以部署到不同的物理节点上。

OpenStack 本身是一个分布式系统，不但各个服务可以分布部署，服务中的组件也可以分布部署。

这种分布式特性让 OpenStack 具备极大的灵活性、伸缩性和高可用性。

当然从另一个角度讲，这也使得 OpenStack 比一般系统复杂，学习难度也更大。

后面章节我们会深入学习 Keystone、Glance、Nova、Neutron 和 Cinder 这几个 OpenStack 最重要最核心的服务。

# 第 4 章

## ◀ 搭建实验环境 ▶

在学习 OpenStack 各服务之前，让我们先搭建起一个实验环境。

毋庸置疑，一个看得到摸得着而且允许我们随便折腾的 OpenStack 能够提高我们的学习效率。

因为是我们自己学习用的实验环境，CloudMan 推荐使用 DevStack。

<http://docs.openstack.org/developer/devstack/>

DevStack 丰富的选项让我们能够灵活地选取和部署想要的 OpenStack 服务，非常适合学习和研究。

## 4.1 部署拓扑

首先我们来设计 OpenStack 的部署拓扑。

OpenStack 是一个分布式系统，由若干不同功能的节点（Node）组成：

- 控制节点（Controller Node）。
- 管理 OpenStack，其上运行的服务有 Keystone、Glance、Horizon 以及 Nova 和 Neutron 中管理相关的组件。
- 控制节点也运行支持 OpenStack 的服务，例如 SQL 数据库（通常是 MySQL）、消息队列（通常是 RabbitMQ）和网络时间服务 NTP。
- 网络节点（Network Node）。
- 其上运行的服务为 Neutron。
- 为 OpenStack 提供 L2 和 L3 网络。
- 包括虚拟机网络、DHCP、路由、NAT 等。
- 存储节点（Storage Node）。
- 提供块存储（Cinder）或对象存储（Swift）服务。
- 计算节点（Compute Node）。
- 其上运行 Hypervisor（默认使用 KVM）。

- 同时运行 Neutron 服务的 agent，为虚拟机提供网络支持。

这几类节点是从功能上进行的逻辑划分，在实际部署时可以根据需求灵活配置，比如：

- 在大规模 OpenStack 生产环境中，每类节点都分别部署在若干台物理服务器上，各司其职并互相协作。这样的环境具备很好的性能、伸缩性和高可用性。
- 在最小的实验环境中，可以将 4 类节点部署到一个物理的甚至是虚拟服务器上。麻雀虽小五脏俱全，通常也称为 All-in-One 部署。

在我们的实验环境中，为了使得拓扑简洁同时功能完备，我们用两个虚拟机(如图 4-1 所示)：

- devstack-controller: 控制节点 + 网络节点 + 块存储节点 + 计算节点。
- devstack-compute: 计算节点。

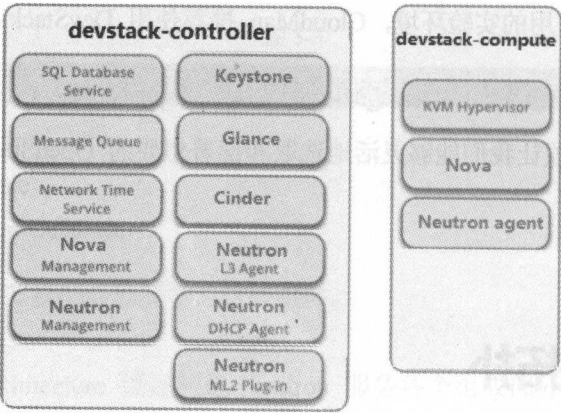


图 4-1

## 4.2 物理资源需求

物理资源中，CPU 和内存配置如图 4-2 所示，供参考。

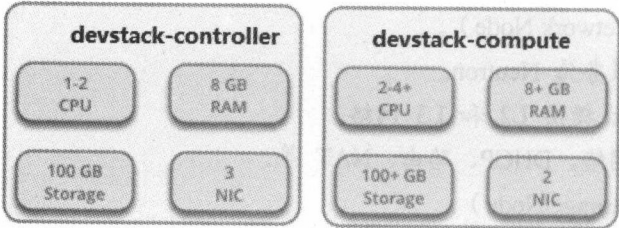


图 4-2

如果是在自己的 PC 机上创建虚拟机部署，资源可能达不到这个要求，可以适当调整。



## 4.3 网络规划

网络部署方面，我们规划了三个网络，如图 4-3 所示。

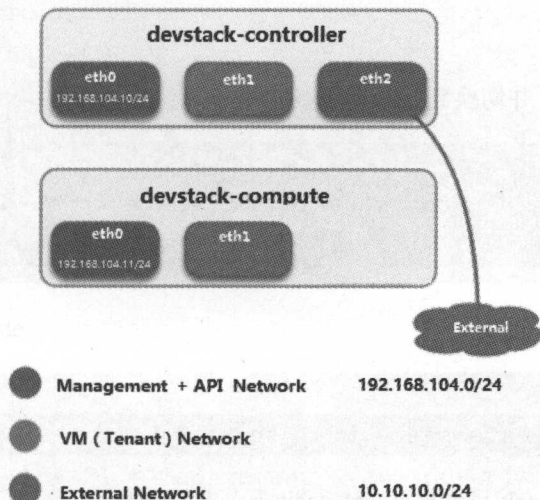


图 4-3

**Management Network:** 用于 OpenStack 内部管理用，比如各服务之间通信。这里使用 eth0。

**VM (Tenant) Network:** OpenStack 部署的虚拟机所使用的网络。OpenStack 支持多租户 (Tenant)，虚拟机是放在 Tenant 下的，所以叫 Tenant Network。这里使用 eth1。

**External Network:** 一般来说，Tenant Network 是内部私有网络，只用于 VM 之间通信，与其他非 VM 网络是隔离的。这里我们规划了一个外部网络 (External Network)，通过 devstack-controller 的 eth2 连接。

Neutron 通过 L3 服务让 VM 能够访问到 External Network。

对于公有云，External Network 一般指的是 Internet。

对于企业私有云，External Network 则可以是 Intranet 中的某个网络。

## 4.4 部署 DevStack

按照以下步骤部署 DevStack。

### 1. 创建虚拟机

按照物理资源需求创建 devstack-controller 和 devstack-compute 虚拟机。

## 2. 安装操作系统

安装 Ubuntu 14.04, 并配置 eth0 的 IP:

- devstack-controller, 192.168.104.10
- devstack-compute, 192.168.104.11

## 3. 下载代码

下载 devstack 代码, 并切换到 stable/liberty 分支。

```
apt-get install git -y
git clone https://git.openstack.org/openstack-dev/devstack -b
stable/liberty
```

## 4. 配置 stack 用户

创建 stack 用户。

```
devstack/tools/create-stack-user.sh
```

为方便起见, 将 devstack 目录放到 /opt/stack 下, 并设置权限。

```
mv devstack /opt/stack
chown -R stack:stack /opt/stack/devstack
```

切换到 stack 用户。

```
su - stack
cd devstack
```

## 5. 编写运行配置文件

在 /opt/stack/devstack 目录下, 创建 local.conf。

### (1) devstack-controller

```
[[local|localrc]]
MULTI_HOST=true
HOST_IP=192.168.104.10 # management & api network
LOGFILE=/opt/stack/logs/stack.sh.log
# Credentials
ADMIN_PASSWORD=admin
MYSQL_PASSWORD=secret
RABBIT_PASSWORD=secret
SERVICE_PASSWORD=secret
```

```

SERVICE_TOKEN=abcdefghijklmnopqrstuvwxyz
# enable neutron-ml2-vlan
disable_service n-net
enable_service
q-svc,q-agt,q-dhcp,q-l3,q-meta,neutron,q-lbaas,q-fwaas,q-vpn
Q_AGENT=linuxbridge
ENABLE_TENANT_VLANS=True
TENANT_VLAN_RANGE=3001:4000
PHYSICAL_NETWORK=default
LOG_COLOR=False
LOGDIR=$DEST/logs
SCREEN_LOGDIR=$LOGDIR/screen

```

## (2) devstack-compute

```

[[local|localrc]]
MULTI_HOST=true
HOST_IP=192.168.104.11 # management & api network
# Credentials
ADMIN_PASSWORD=admin
MYSQL_PASSWORD=secret
RABBIT_PASSWORD=secret
SERVICE_PASSWORD=secret
SERVICE_TOKEN=abcdefghijklmnopqrstuvwxyz
# Service information
SERVICE_HOST=192.168.104.10
MYSQL_HOST=$SERVICE_HOST
RABBIT_HOST=$SERVICE_HOST
GLANCE_HOSTPORT=$SERVICE_HOST:9292
Q_HOST=$SERVICE_HOST
KEYSTONE_AUTH_HOST=$SERVICE_HOST
KEYSTONE_SERVICE_HOST=$SERVICE_HOST
CEILOMETER_BACKEND=mongodb
DATABASE_TYPE=mysql
ENABLED_SERVICES=n-cpu,q-agt,neutron
Q_AGENT=linuxbridge
ENABLE_TENANT_VLANS=True
TENANT_VLAN_RANGE=3001:4000
PHYSICAL_NETWORK=default
# vnc config

```



```
NOVA_VNC_ENABLED=True
NOVNC_PROXY_URL="http://$SERVICE_HOST:6080/vnc_auto.html"
VNC_SERVER_LISTEN=$HOST_IP
VNC_SERVER_PROXYCLIENT_ADDRESS=$VNC_SERVER_LISTEN
LOG_COLOR=False
LOGDIR=$DEST/logs
SCREEN_LOGDIR=$LOGDIR/screen
```

另外，为了加快安装速度，还可以加上下面的配置使用国内的 devstack 镜像站点。

```
# use TryStack git mirror
GIT_BASE=http://git.trystack.cn
NOVNC_REPO=http://git.trystack.cn/kanaka/noVNC.git
SPICE_REPO=http://git.trystack.cn/git/spice/spice-html5.git
```

## 6. 开始部署

分别在 devstack-controller 和 devstack-compute 上执行命令。

```
./stack.sh
```

会输出各项操作的结果。

日志会写到 stack.sh.log 文件。

整个过程需要连接 Internet，网速慢可能会花较长时间，成功后最后会打印出相关信息。

devstack-controller 上的输出：

```
This is your host IP address: 192.168.104.10
This is your host IPv6 address: ::1
Horizon is now available at http://192.168.104.10/
Keystone is serving at http://192.168.104.10:5000/
The default users are: admin and demo
The password: admin
```

devstack-compute 上的输出：

```
This is your host IP address: 192.168.104.11
This is your host IPv6 address: ::1
stack.sh completed in 192 seconds.
```

## 7. 验证 OpenStack

下面验证 OpenStack 已经正常运行。

通过浏览器访问 <http://192.168.104.10/> (devstack-controller 的 IP)。

使用 admin/admin 登录，如图 4-4 所示。



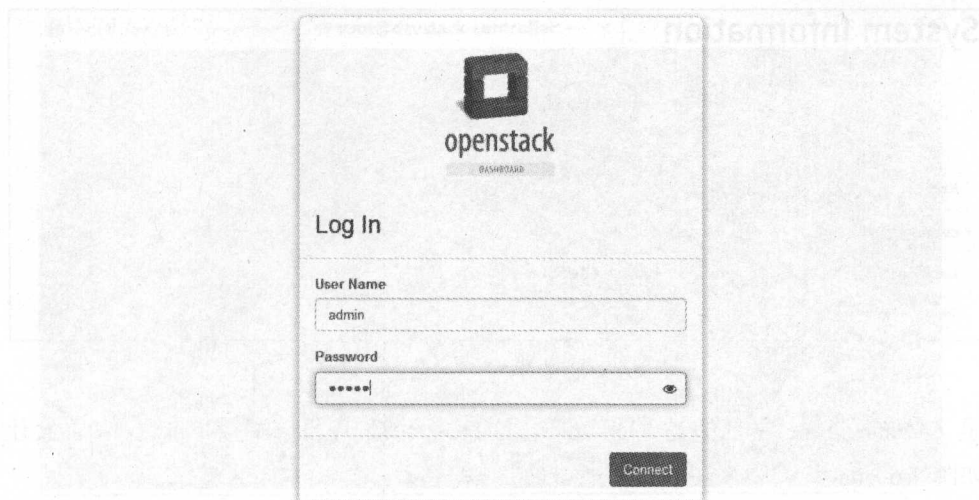


图 4-4

单击 System → System Information，确保各个服务的状态正常，如图 4-5 所示。

Services    Compute Services    Block Storage Services    Network Agents			
Filter			
Name	Service	Host	Status
nova	compute	192.168.104.10	Enabled
neutron	network	192.168.104.10	Enabled
cinderv2	volumev2	192.168.104.10	Enabled
glance	image	192.168.104.10	Enabled
nova_legacy	compute_legacy	192.168.104.10	Enabled
cinder	volume	192.168.104.10	Enabled
ec2	ec2	192.168.104.10	Enabled
keystone	identity (native backend)	192.168.104.10	Enabled
Displaying 8 items			

Services    Compute Services    Block Storage Services    Network Agents					
Filter					
Name	Host	Zone	Status	State	Last Updated
nova-conductor	devstack-controller	internal	Enabled	Up	0 minutes
nova-cert	devstack-controller	internal	Enabled	Up	0 minutes
nova-scheduler	devstack-controller	internal	Enabled	Up	0 minutes
nova-consoleauth	devstack-controller	internal	Enabled	Up	0 minutes
nova-compute	devstack-controller	nova	Enabled	Up	0 minutes
nova-compute	devstack-compute1	nova	Enabled	Up	0 minutes
Displaying 6 items					

System Information				
Services   Compute Services   Block Storage Services   Network Agents				
Name	Host	Zone	Status	State
cinder-scheduler	devstack-controller	nova	Enabled	Up
cinder-volume	devstack-controller@lvmdriver-1	nova	Enabled	Down
Displaying 2 items				

图 4-5

这里 `cinder-volume` 的状态是“Down”，这是因为我们还没有配置 `cinder`，后面会让它 Up 起来，如图 4-6 所示。

System Information				
Services   Compute Services   Block Storage Services   Network Agents				
Type	Name	Host	Status	State
Metadata agent	neutron-metadata-agent	devstack-controller	Enabled	Up
DHCP agent	neutron-dhcp-agent	devstack-controller	Enabled	Up
Linux bridge agent	neutron-linuxbridge-agent	devstack-compute1	Enabled	Up
L3 agent	neutron-l3-agent	devstack-controller	Enabled	Up
Linux bridge agent	neutron-linuxbridge-agent	devstack-controller	Enabled	Up
Displaying 5 items				

图 4-6

## 8. 启动 OpenStack

如果重启了系统，OpenStack 不会自动启动，可以运行下面命令：

```
cd devstack
./rejoin-stack.sh
```

运行成功后，OpenStack 的每个服务都在一个 `screen` 中以进程方式运行。

使用 `screen` 有几个好处：

(1) 可以方便地在不同服务之间切换和查看日志。

因为 OpenStack 的服务很多，每个服务都有自己的日志文件，查找日志是一件非常麻烦的事情，使用 `screen` 可以帮我们提高效率。

(2) 当我们修改了某个服务的配置文件需要重启服务时，只需在该服务的 `screen` 窗口按 `Ctrl+C` 键，然后在命令行中找到上一个命令（Up 键）执行就行，这个命令就是启动服务的命令，如图 4-7 所示。

```

root@devstack-controller: ~
root@devstack-controller: ~ x
on2.7/dist-packages/oslo_messaging/drivers/ampq.py:103
2015-10-10 22:00:09.496 INFO oslo_messaging.drivers.impl_rabbit [req-6fd7836a-b
2015-10-10 22:00:09.049 INFO oslo_messaging.drivers.impl_rabbit [req-6fd7836a-b
2015-10-10 22:00:09.056 INFO neutron.plugins.ml2.drivers.linuxbridge.agent.linux
gent RPC Daemon Started!
2015-10-10 22:00:09.128 DEBUG neutron.agent.linux.utils [req-6fd7836a-b4fd-412e-
ute_rootwrap_daemon /bpt/stack/neutron/neutron/agent/linux/utils.py:101
2015-10-10 22:00:09.179 DEBUG oslo_rootwrap.client [req-6fd7836a-b4fd-412e-8c3e-
etc/neutron/rootwrap.conf] command has been instantiated, initialize /usr/local
2015-10-10 22:00:09.181 7018 DEBUG oslo_messaging.drivers.ampq [-] Pool creatin
ampq.py:103
2015-10-10 22:00:09.183 7018 INFO oslo_messaging.drivers.impl_rabbit [-] connec
2015-10-10 22:00:10.105 7018 INFO oslo_messaging.drivers.impl_rabbit [-] connec
2015-10-10 22:00:10.114 7018 DEBUG oslo_service.loopingcall [-] Fixed interval l
ml2.drivers.linuxbridge.agent.linuxbridge_neutron_agent.LinuxBridgeNeutronAgentR
n2.7/dist-packages/oslo_service/loopingcall.py:117
2015-10-10 22:00:55.676 7018 DEBUG oslo_service.loopingcall [-] Fixed interval l
oping call <bound method LinuxBridgeNeutronAgentRPC._report_state of <neutron.p
ugins.ml2.drivers.linuxbridge.agent.linuxbridge_neutron_agent.LinuxBridgeNeuro
nAgentRPC object at 0x7f86dd93c150>> sleeping for 13.51 seconds _run_loop /usr/l
ocal/lib/python2.7/dist-packages/oslo_service/loopingcall.py:117
.85(L) q-agt* 95(L) q-dhcp 105(L) q-l3 115(L) q-meta 125(L) n-cond 135(L) n
crt 145(L) n-sch 155(L) n-novnc 165(L) n-cauth 175(L) n-cpu 185(L) c-api
195(L)

```

图 4-7

下面是 screen 常用命令。

在 screen 中执行：

- Ctrl+a+n，切换到下一个窗口。
- Ctrl+a+p，切换到前一个窗口（与 Ctrl+a+n 相对）。
- Ctrl+a/0~9，切换到窗口 0~9。
- Ctrl+a+d，暂时断开（detach）当前 screen 会话，但不中断 screen 窗口中程序的运行。

在 shell 中执行：

- screen -ls，列出当前所有的 session。
- screen -r stack，回到 devstack 这个 session。

## 9. 删除自动创建的网络

Devstack 在部署时可能会创建几个测试网络，为了得到一个干净的环境需将其删除。

### （1）删除 Router

Admin → System → Routers，如图 4-8 所示操作。

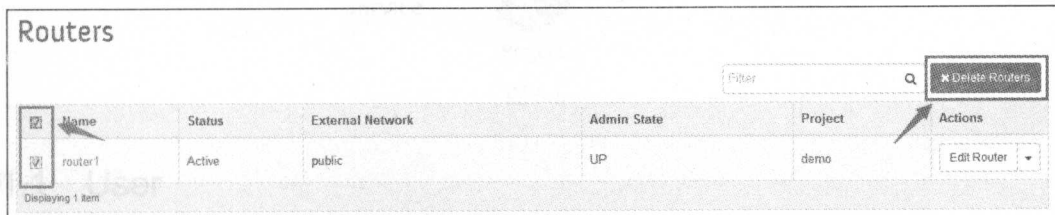


图 4-8

### （2）删除 Network

Admin → System → Networks，如图 4-9 所示操作。

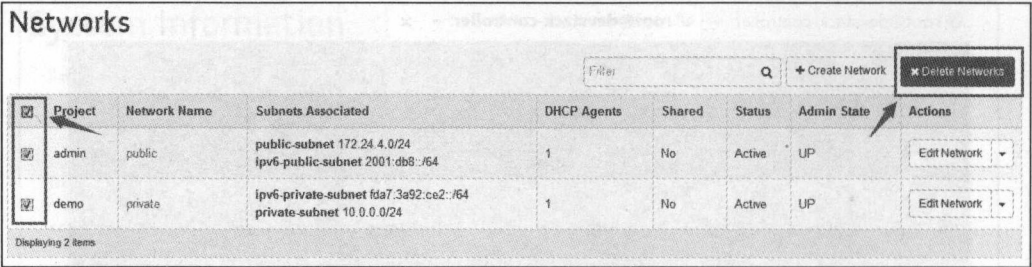


图 4-9

至此，我们得到了一个干净的 OpenStack 环境。  
后面我们将在这个环境中实践各个模块。



# 第 5 章

## Identity Service——Keystone

作为 OpenStack 的基础支持服务，Keystone 完成下面这几件事情：

- 管理用户及其权限。
- 维护 OpenStack Services 的 Endpoint。
- Authentication（认证）和 Authorization（鉴权）。

### 5.1 概念

学习 Keystone，得理解如图 5-1 所示的这些概念：User、Role、Credentials、Authentication、Endpoint、Service、Project、Token。

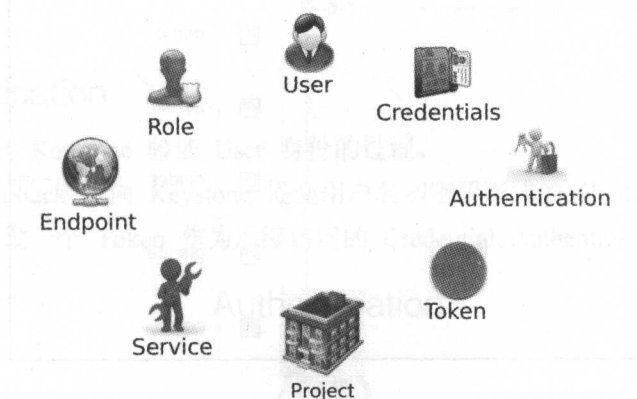


图 5-1

#### 5.1.1 User

User 指代任何使用 OpenStack 的实体，可以是真正的用户，其他系统或者服务。User 图标如图 5-2 所示。

# USER

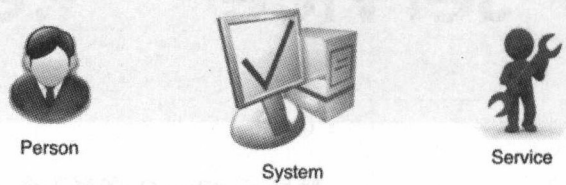


图 5-2

当 User 请求访问 OpenStack 时，Keystone 会对其进行验证。  
Horizon 在 Identity → Users 管理 User，如图 5-3 所示。

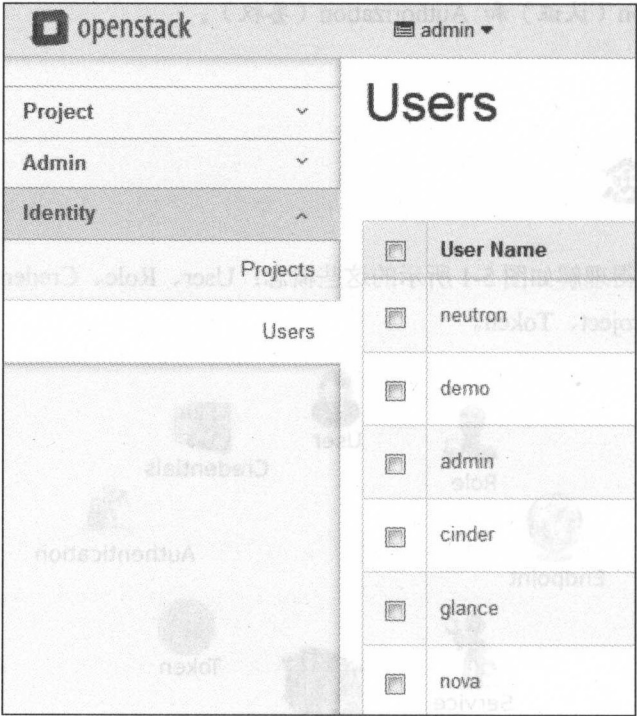


图 5-3

除了 admin 和 demo，OpenStack 也为 nova、cinder、glance、neutron 服务创建了相应的 User。

admin 也可以管理这些 User，如图 5-4 所示。

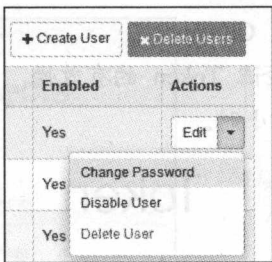


图 5-4

### 5.1.2 Credentials

Credentials 是 User 用来证明自己身份的信息，可以是：

- (1) 用户名/密码
- (2) Token
- (3) API Key
- (4) 其他高级方式

Credentials 图标如图 5-5 所示。

Credentials



图 5-5

### 5.1.3 Authentication

Authentication 是 Keystone 验证 User 身份的过程。

User 访问 OpenStack 时向 Keystone 提交用户名和密码形式的 Credentials，Keystone 验证通过后会给 User 签发一个 Token 作为后续访问的 Credential。Authentication 图标如图 5-6 所示。

Authentication



图 5-6

### 5.1.4 Token

Token 是由数字和字母组成的字符串，User 成功 Authentication 后，它由 Keystone 分配给 User，如图 5-7 所示。

- Token 用做访问 Service 的 Credential。
- Service 会通过 Keystone 验证 Token 的有效性。
- Token 的有效期默认是 24 小时。



图 5-7

## 5.1.5 Project

Project 用于将 OpenStack 的资源（计算、存储和网络）进行分组和隔离。

根据 OpenStack 服务的对象不同，Project 可以是一个客户（公有云，也叫租户）、部门或者项目组（私有云）。

这里请注意：

- 资源的所有权是属于 Project 的，而不是 User。
- 在 OpenStack 的界面和文档中，Tenant / Project / Account 这几个术语是通用的，但长期看会倾向使用 Project。
- 每个 User（包括 admin）必须挂在 Project 里才能访问该 Project 的资源。一个 User 可以属于多个 Project。
- admin 相当于 root 用户，具有最高权限。

Horizon 在 Identity → Projects 中管理 Project，Project 图标如图 5-8 所示。



图 5-8

Project 界面如图 5-9 所示。

通过 Manage Members 将 User 添加到 Project 中，如图 5-10 和图 5-11 所示。



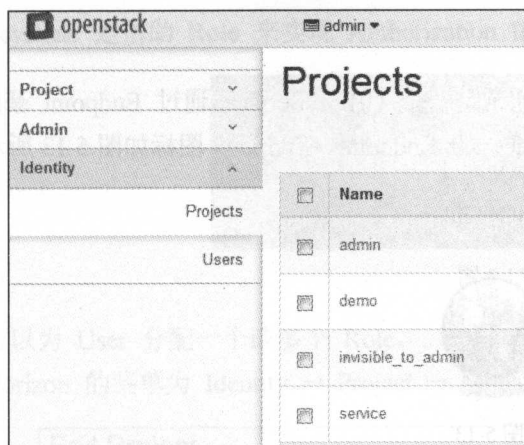


图 5-9

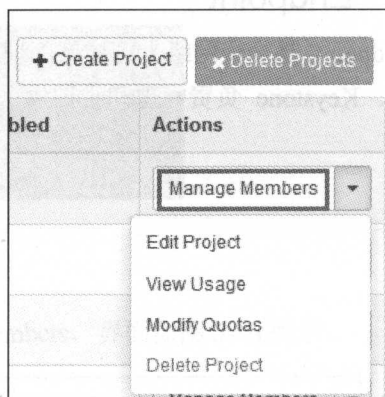


图 5-10

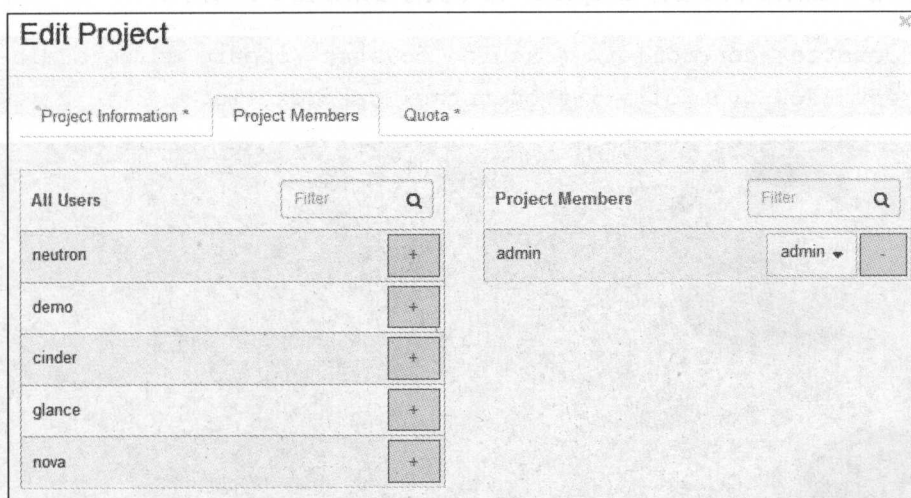


图 5-11

### 5.1.6 Service

OpenStack 的 Service 包括 Compute (Nova)、Block Storage (Cinder)、Object Storage (Swift)、Image Service (Glance)、Networking Service (Neutron) 等。

每个 Service 都会提供若干个 Endpoint, User 通过 Endpoint 访问资源和执行操作。图标如图 5-12 所示。

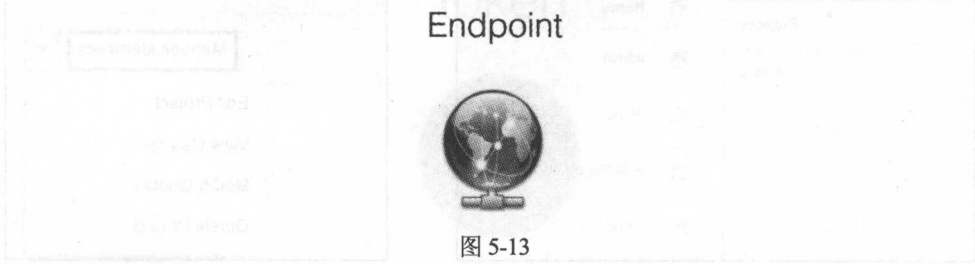
#### Service



图 5-12

### 5.1.7 Endpoint

Endpoint 是一个网络上可访问的地址，通常是一个 URL。Service 通过 Endpoint 暴露自己的 API。Keystone 负责管理和维护每个 Service 的 Endpoint。Endpoint 图标如图 5-13 所示。



可以使用下面的命令来查看 Endpoint。命令执行结果如图 5-14 所示。

```
root@devstack-controller:~# source devstack/openrc admin admin
root@devstack-controller:~# openstack catalog list
```

```
root@devstack-controller:~# openstack catalog list
```

Name	Type	Endpoints
nova	compute	RegionOne publicURL: http://192.168.111.22:8774/v2.1/0d1950f32fca4138b67c2b1049b8b3dd internalURL: http://192.168.111.22:8774/v2.1/0d1950f32fca4138b67c2b1049b8b3dd adminURL: http://192.168.111.22:8774/v2.1/0d1950f32fca4138b67c2b1049b8b3dd
neutron	network	RegionOne publicURL: http://192.168.111.22:9696 internalURL: http://192.168.111.22:9696/ adminURL: http://192.168.111.22:9696/
cinderv2	volumev2	RegionOne publicURL: http://192.168.111.22:8776/v2/0d1950f32fca4138b67c2b1049b8b3dd internalURL: http://192.168.111.22:8776/v2/0d1950f32fca4138b67c2b1049b8b3dd adminURL: http://192.168.111.22:8776/v2/0d1950f32fca4138b67c2b1049b8b3dd
glance	image	RegionOne publicURL: http://192.168.111.22:9292 internalURL: http://192.168.111.22:9292 adminURL: http://192.168.111.22:9292
nova_legacy	compute_legacy	RegionOne publicURL: http://192.168.111.22:8774/v2/0d1950f32fca4138b67c2b1049b8b3dd internalURL: http://192.168.111.22:8774/v2/0d1950f32fca4138b67c2b1049b8b3dd adminURL: http://192.168.111.22:8774/v2/0d1950f32fca4138b67c2b1049b8b3dd
cinder	volume	RegionOne publicURL: http://192.168.111.22:8776/v1/0d1950f32fca4138b67c2b1049b8b3dd internalURL: http://192.168.111.22:8776/v1/0d1950f32fca4138b67c2b1049b8b3dd adminURL: http://192.168.111.22:8776/v1/0d1950f32fca4138b67c2b1049b8b3dd
ec2	ec2	RegionOne publicURL: http://192.168.111.22:8773 internalURL: http://192.168.111.22:8773/ adminURL: http://192.168.111.22:8773/
keystone	identity	RegionOne publicURL: http://192.168.111.22:5000/v2.0 internalURL: http://192.168.111.22:5000/v2.0 adminURL: http://192.168.111.22:35357/v2.0/

图 5-14

### 5.1.8 Role

安全包含两部分：Authentication（认证）和 Authorization（鉴权）。

- Authentication 解决的是“你是谁？”的问题。
- Authorization 解决的是“你能干什么？”的问题。

Keystone 是借助 Role 来实现 Authorization 的，Keystone 定义 Role，如图 5-15 所示。

```
root@devstack-controller:~# openstack role list
+-----+-----+
| ID                                           | Name |
+-----+-----+
| 09441190f690466da2d49753811e933a         |      |
| 68170e25faa54bafad44bf08c9e68a88         | anotherrole |
| 6c8fa1b93c594d5c953cc62958d18759         | ResellerAdmin |
| afc914c54392467d9137e04859f7ded9         | admin |
| bdb78ba464c5468484c5d56baf2301e2         | service |
|                                             | Member |
+-----+-----+
```

图 5-15

可以为 User 分配一个或多个 Role。

Horizon 的菜单为 Identity → Project → Manage Members，界面如图 5-16 所示。

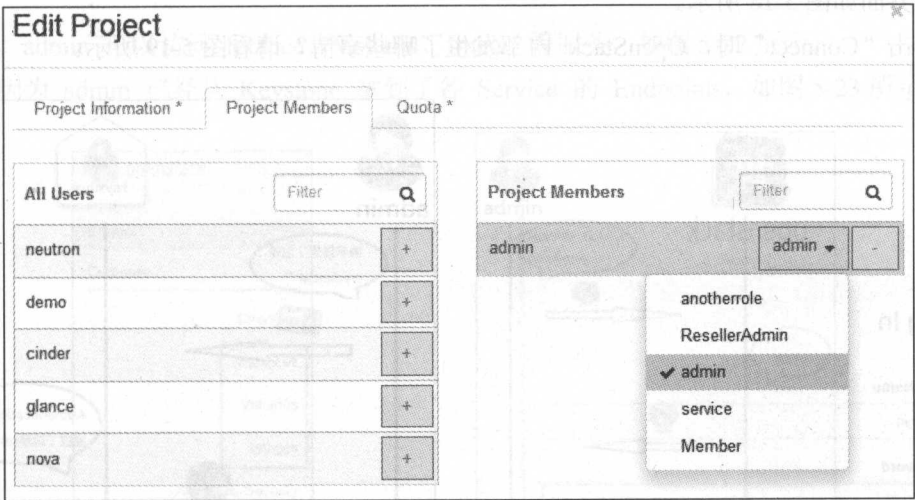


图 5-16

Service 决定每个 Role 能做什么事情。

Service 通过各自的 policy.json 文件对 Role 进行访问控制。

下面是 Nova 服务 /etc/nova/policy.json 中的示例，如图 5-17 所示。



图 5-17

上面配置的含义是：对于 create、attach\_network 和 attach\_volume 操作，任何 Role 的 User 都可以执行，但只有 admin 这个 Role 的 User 才能执行 forced\_host 操作。

OpenStack 默认配置只区分 admin 和非 admin Role。如果需要对特定的 Role 进行授权，可以修改 policy.json。

## 5.2 通过例子学习

Keystone 的相关概念介绍完了，下面通过“查询可用 image”这个实际操作，让大家对这些概念建立更加感性的认识。

假定 User admin 要查看 Project 中的 image。

### 5.2.1 第 1 步 登录

登录界面如图 5-18 所示。

当单击“Connect”时，OpenStack 内部发生了哪些事情？请看图 5-19 所示。



图 5-18

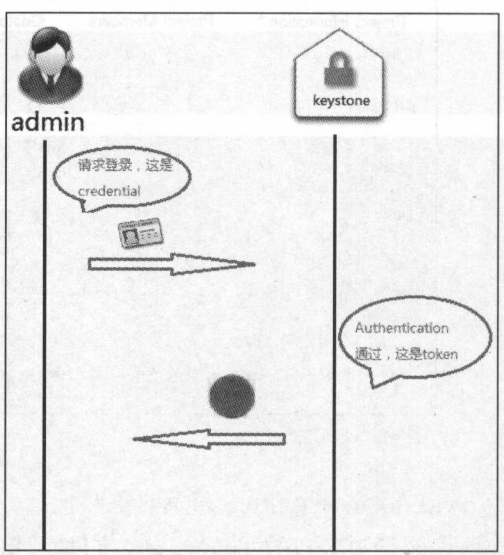


图 5-19

Token 中包含了 User 的 Role 信息。

### 5.2.2 第 2 步 显示操作界面

操作界面如图 5-20 所示。请注意，顶部显示 admin 可访问的 Project 为 admin 和 demo。其实在此之前发生了一些事情，如图 5-21 所示。



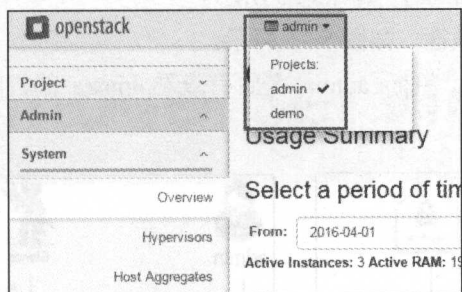


图 5-20

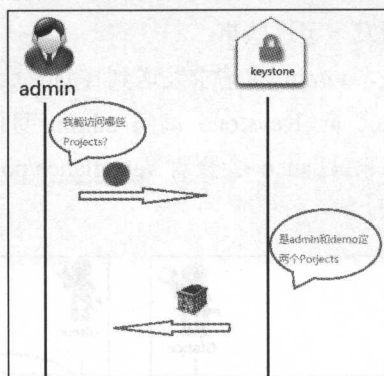


图 5-21

同时, admin 可以访问 Instance、Volume、Image 等服务, 如图 5-22 所示。

这是因为 admin 已经从 Keystone 拿到了各 Service 的 Endpoints, 如图 5-23 所示。

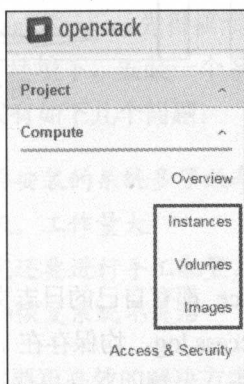


图 5-22

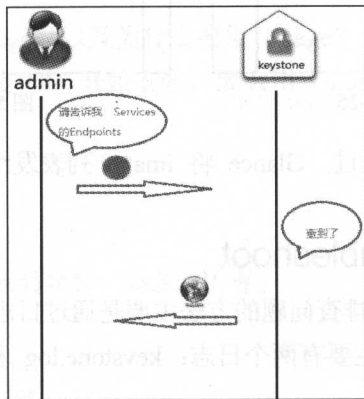


图 5-23

### 5.2.3 第 3 步 显示 image 列表

单击“Images”，会显示 image 列表，如图 5-24 所示。

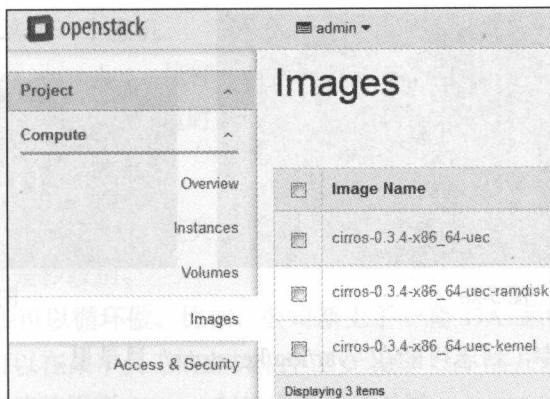


图 5-24

背后发生了这些事：

首先，admin 将请求发送到 Glance 的 Endpoint，如图 5-25 所示。

Glance 向 Keystone 询问 admin 身份的有效性，如图 5-26 所示。

接下来 Glance 会查看 /etc/glance/policy.json，判断 admin 是否有查看 image 的权限，如图 5-27 所示。

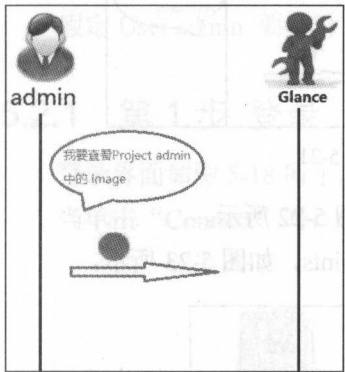


图 5-25

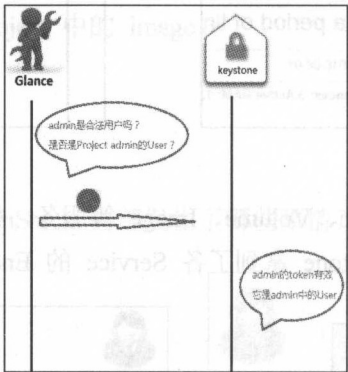


图 5-26

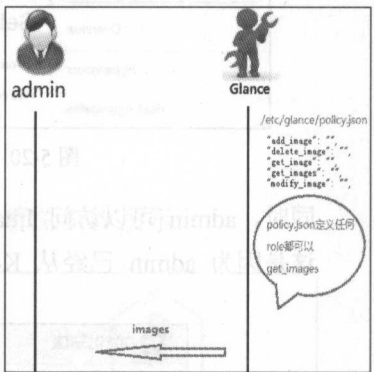


图 5-27

权限判定通过，Glance 将 image 列表发给 admin。

### 5.2.4 Troubleshoot

OpenStack 排查问题的方法主要是通过日志。每个 Service 都有自己的日志文件。

Keystone 主要有两个日志：keystone.log 和 keystone\_access.log，均保存在 /var/log/apache2/ 目录里。

devstack 的 screen 窗口已经帮我们打开了这两个日志。

如图 5-28 所示，可以直接查看日志信息。

如果需要得到最详细的日志信息，可以在 /etc/keystone/keystone.conf 中打开 debug 选项，如图 5-29 所示。

```
tonetoken (audit_id=G1x1v1xFRZuB2BK-U22PGA, au
one) process_request /opt/stack/keystone/keyst
2015-10-13 15:26:24.373957 3318 INFO keystone.
2015-10-13 15:26:24.374149 3318 DEBUG keystone.
d_policy_check_credentials /opt/stack/keystone
2015-10-13 15:26:24.374305 3318 DEBUG keystone.
ment_build_policy_check_credentials /opt/stac
2015-10-13 15:26:24.383472 3318 DEBUG keystone.
ed_auth': False, 'access_token_id': None, 'use
one, 'token': <KeystoneToken (audit_id=G1x1v1x
3bb, 'trust_id': None)} enforce /opt/stack/key
2015-10-13 15:26:24.385079 3318 DEBUG keystone
tone/keystone/common/controller.py:162
13(L) key* 25(L) key-access 35(L) horizon
```

图 5-28

```
[DEFAULT]
max_token_size = 16384
logging_exception_prefix = %(p
logging_debug_format_suffix =
logging_default_format_string
logging_context_format_string
debug = True
admin_token = abcdefghijklmnop
rpc_backend = rabbit
```

图 5-29

在非 devstack 安装中，日志可能在 /var/log/keystone/ 目录里。

# 第 6 章

## ◀ Image Service——Glance ▶

### 6.1 理解 Image

要理解 Image Service 先得搞清楚什么是 Image，以及为什么要用 Image？

在传统 IT 环境下，安装一个系统要么从安装 CD 开始安装，要么用 Ghost 等克隆工具恢复。这两种方式有如下几个问题：

- 如果要安装的系统多了效率就很低
- 时间长，工作量大
- 安装完还要进行手工配置，比如安装其他的软件，设置 IP 等
- 备份和恢复系统不灵活

云环境下需要更高效的解决方案，这就是 Image。

Image 是一个模板，里面包含了基本的操作系统和其他的软件。

举例来说，有家公司需要为每位员工配置一套办公用的系统，一般需要一个 Windows 7 系统再加 MS office 软件。

OpenStack 是这么玩的：

- (1) 先手工安装好这么一个虚拟机。
- (2) 然后对虚拟机执行 snapshot，这样就得到了一个 image。
- (3) 当有新员工入职需要办公环境时，立马启动一个或多个该 image 的 instance（虚拟机）就可以了。

在这个过程中，第 1 步跟传统方式类似，需要手工操作和一定时间，但第 2 步、第 3 步非常快，全自动化，一般都是秒级别。

而且第 2 步、第 3 步可以循环做。比如，公司新上了一套 OA 系统，每个员工的 PC 上都有客户端软件，那么可以在某个员工的虚拟机中手工安装好 OA 客户端，然后执行 snapshot，得到新的 image，以后直接使用新 image 创建虚拟机就可以了。

另外，snapshot 还有备份的作用，能够非常方便地恢复系统。

## 6.2 理解 Image Service

Image Service 的功能是管理 Image，让用户能够发现、获取和保存 Image。在 OpenStack 中，提供 Image Service 的是 Glance，其具体功能如下：

- 提供 REST API，让用户能够查询和获取 image 的元数据和 image 本身。
- 支持多种方式存储 image，包括普通的文件系统、Swift、Amazon S3 等。
- 对 Instance 执行 Snapshot 创建新的 image。

## 6.3 Glance 架构

Glance 的架构如图 6-1 所示。

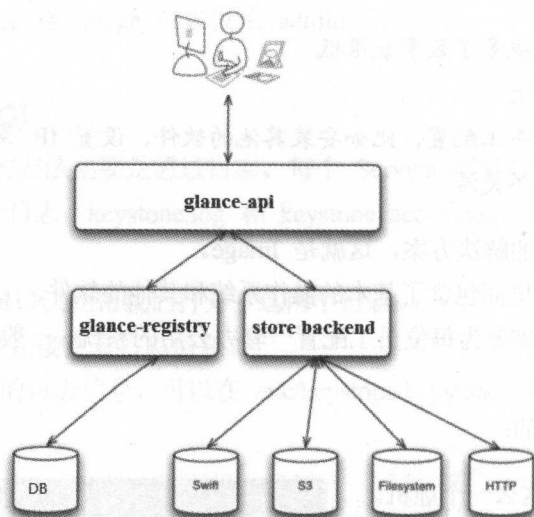


图 6-1

从图 6-1 可以看到，Glance 主要由三部分组成。

### 1. glance-api

glance-api 是系统后台运行的服务进程，对外提供 REST API，响应 image 查询、获取和存储的调用，glance-api 不会真正处理请求。

如果是与 image metadata（元数据）相关的操作，glance-api 会把请求转发给 glance-registry；如果是与 image 自身存取相关的操作，glance-api 会把请求转发给该 image 的 store backend。

在控制节点上可以查看 glance-api 进程，如图 6-2 所示。



```
root@devstack-controller:~# ps -e|grep glance-api
4131 pts/10    00:16:47  glance-api
4335 pts/10    00:09:48  glance-api
4337 pts/10    00:00:01  glance-api
```

图 6-2

2. glance-registry

glance-registry 是系统后台运行的服务进程，负责处理和存取 image 的 metadata，例如 image 的大小和类型。

在控制节点上可以查看 glance-registry 进程，如图 6-3 所示。

```
root@devstack-controller:~# ps -e|grep glance-registry
4146 pts/9      00:00:01  glance-registry
4314 pts/9      00:00:05  glance-registry
4315 pts/9      00:00:01  glance-registry
```

图 6-3

Glance 支持多种格式的 image，包括如图 6-4 所示的文件格式。

<b>Raw</b>	This is an unstructured disk image format
<b>vhd</b>	This is the VHD disk format, a common disk format used by virtual machine monitors from VMWare,
<b>vmdk</b>	Another common disk format supported by many common virtual machine monitors
<b>VDI</b>	A disk format supported by VirtualBox virtual machine monitor and the QEMU emulator
<b>ISO</b>	An archive format for the data contents of an optical disc (e.g. CDROM)
<b>QCOW2</b>	A disk format supported by the QEMU emulator that can expand dynamically and supports Copy on Write
<b>aki</b>	This indicates what is stored in Glance is an Amazon kernel image
<b>ari</b>	This indicates what is stored in Glance is an Amazon ramdisk image
<b>ami</b>	This indicates what is stored in Glance is an Amazon machine image

图 6-4

3. Database

Image 的 metadata 会保持到 database 中，默认是 MySQL。

在控制节点上可以查看 glance 的 database 信息，如图 6-5 所示。

```

root@devstack-controller:~# su - stack
stack@devstack-controller:~$ mysql
Welcome to the MySQL monitor.  Commands end with ; or \g
Your MySQL connection id is 490
Server version: 5.5.44-0ubuntu0.14.04.1 (Ubuntu)

Copyright (c) 2000, 2015, Oracle and/or its affiliates
Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use glance
Reading table information for completion of table and column names
You can turn off this feature to get a quicker start with MySQL.
Database changed
mysql> show tables;
+-----+
| Tables_in_glance |
+-----+
| artifact_blob_locations |
| artifact_blobs |
| artifact_dependencies |
| artifact_properties |
| artifact_tags |
| artifacts |
| image_locations |
| image_members |
| image_properties |
| image_tags |
| images |
| metadef_namespace_resource_types |
| metadef_namespaces |
| metadef_objects |
| metadef_properties |
| metadef_resource_types |
| metadef_tags |
| migrate_version |
| task_info |
| tasks |
+-----+
20 rows in set (0.00 sec)

```

图 6-5

#### 4. Store backend

Glance 自己并不存储 image，真正的 image 是存放在 backend 中的。

Glance 支持多种 backend，包括：

- A directory on a local file system（这是默认配置）
- GridFS
- Ceph RBD
- Amazon S3
- Sheepdog
- OpenStack Block Storage (Cinder)
- OpenStack Object Storage (Swift)
- VMware ESX

具体使用哪种 backend，是在 `/etc/glance/glance-api.conf` 中配置的。

在我们的 devstack 环境中，image 存放在控制节点本地目录 /opt/stack/data/glance/images/ 中，配置信息如图 6-6 所示。

```
[glance_store]
filesystem_store_datadir = /opt/stack/data/glance/images/
```

图 6-6

其他 backend 的配置可参考这个网址：

<http://docs.openstack.org/liberty/config-reference/content/configuring-glance-image-service-backends.html>

查看目前已经存在的 image，命令及其执行结果如图 6-7 所示。

```
stack@devstack-controller:~$ source devstack/openrc admin admin
stack@devstack-controller:~$ glance image-list
```

ID	Name
90235f54-475c-4c56-b1a5-0fff5ff57161	cirros-0.3.4-x86_64-uec
3ddc5a7e-ce62-4c65-a694-d949b530c08b	cirros-0.3.4-x86_64-uec-kernel
97bd1ddc-bcc9-415f-b42c-e460a6ef0669	cirros-0.3.4-x86_64-uec-ramdisk

图 6-7

查看保存目录，命令及其执行结果如图 6-8 所示。

```
stack@devstack-controller:~$ ls -l /opt/stack/data/glance/images/
total 33096
-rw-r----- 1 stack stack 4979632 Oct 12 07:43 3ddc5a7e-ce62-4c65-a694-d949b530c08b
-rw-r----- 1 stack stack 25165824 Oct 12 07:43 90235f54-475c-4c56-b1a5-0fff5ff57161
-rw-r----- 1 stack stack 3740163 Oct 12 07:43 97bd1ddc-bcc9-415f-b42c-e460a6ef0669
```

图 6-8

每个 image 在目录下都对应有一个文件，文件以 image 的 ID 命名。

## 6.4 Glance 操作

本节将讨论 Glance 的主要操作。

OpenStack 为终端用户提供了 Web UI (Horizon) 和命令行 CLI 两种交换界面，两种方式我们都要会用。可能有些同学觉得既然有更友好的 Web UI 了，为什么还要用 CLI？

这里 CloudMan 给出下面的理由：

- Web UI 的功能没有 CLI 全，有些操作只提供了 CLI。即便是都有的功能，CLI 可以使用的参数更多。
- 一般来说，CLI 返回结果更快，操作起来更高效。
- CLI 可放在脚本中进行批处理。

- 有些耗时的操作 CLI 更合适，比如创建镜像（后面将涉及）。

### 6.4.1 创建 image

#### 1. Web UI

admin 登录后，单击 Project → Compute → Images，界面如图 6-9 所示。

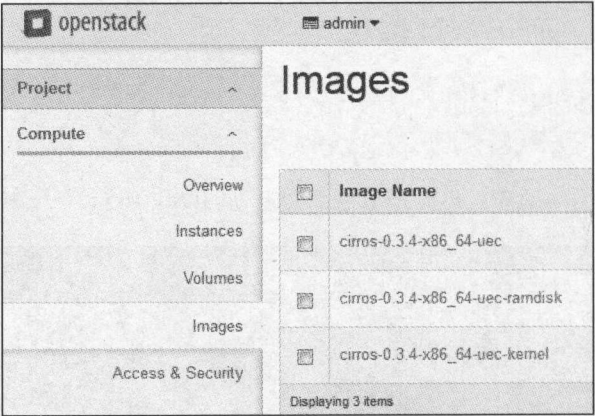


图 6-9

单击右上角“Create Image”按钮，为新 image 命名。如图 6-10 所示。

Create An Image

Name \*

cirros

Description

cirros image

Image Source

Image File

Image File ?

浏览...

cirros-0.3.4-x86\_64-disk.img

图 6-10

这里我们上传一个 image。

单击“浏览”，选择镜像文件 cirros-0.3.4-x86\_64-disk.img。这个 cirros 是一个很小的 linux 镜像，非常适合测试用。大家可以到 <http://download.cirros-cloud.net/> 下载。

格式选择 QCOW2，如图 6-11 所示。



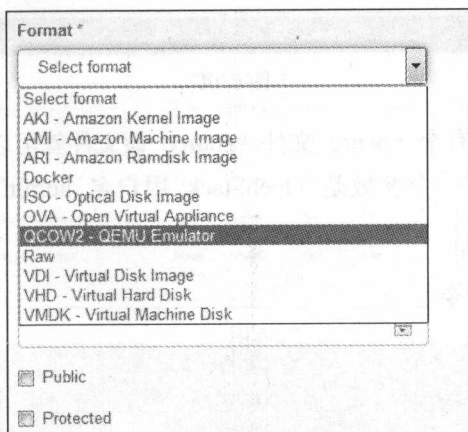


图 6-11

如果选中“Public”，该 image 可以被其他 Project 使用，如果选中“Protected”，该 image 不允许被删除。

单击“Create Image”，文件上传到 OpenStack 并创建新的 image，如图 6-12 所示。

单击 image 链接“cirros”，显示详细信息，如图 6-13 所示。

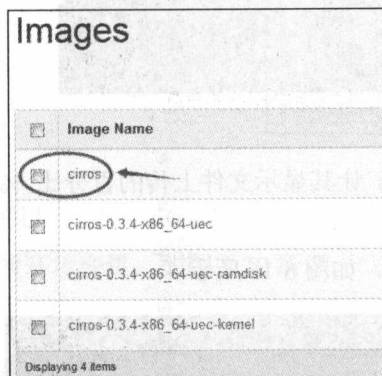


图 6-12

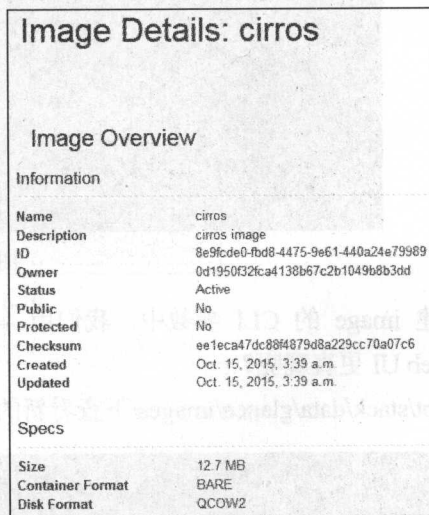


图 6-13

## 2. CLI

cirros 这个 Linux 镜像很小，通过 Web UI 上传很快，操作会很顺畅。

但如果我们要上传的镜像比较大（比如好几个 GB），那么，操作会长时间停留在上传的 Web 界面，我们也不知道目前到底处于什么状态。

对于这样情形下的操作，CLI 是更好的选择。

- 将 image 上传到控制节点的文件系统中，例如 /tmp/cirros-0.3.4-x86\_64-disk.img，设置环境变量，如图 6-14 所示。

```
stack@devstack-controller:~$ source devstack/openrc admin admin
```

图 6-14

Devstack 的安装目录下有个 openrc 文件。source 该文件就可以配置 CLI 的环境变量。这里我们传入了两个参数，第一个参数是 OpenStack 用户名 admin，第二个参数是 Project 名 admin。

- 执行 image 创建命令

```
glance image-create --name cirros --file
/tmp/cirros-0.3.4-x86_64-disk.img --disk-format qcow2 --container-format
bare --progress
```

命令执行结果如图 6-15 所示。

```
stack@devstack-controller:~$ glance image-create --name cirros --file /tmp/cirros-0.3.4-x86_64-disk.img --disk-format qcow2 --container-format bare --progress
[=====] 100%
Property      value
-----
checksum      ee1eca47dc88f4879d8a229cc70a07c6
container_format bare
created_at    2015-10-15T03:59:08Z
disk_format   qcow2
id            2e281bf5-daf6-4625-b7c8-edb2e17c2926
min_disk      0
min_ram       0
name          cirros
owner         0d1950f32fca4138b67c2b1049b8b3dd
protected     False
size          13287936
status        active
tags          []
updated_at    2015-10-15T03:59:08Z
virtual_size   None
visibility     private
```

图 6-15

在创建 image 的 CLI 参数中，我们用 --progress 让其显示文件上传的百分比 %，这样是不是比 Web UI 更直观呢？

在 /opt/stack/data/glance/images/ 下查看新的 Image，如图 6-16 所示。

```
stack@devstack-controller:~$ ls -l /opt/stack/data/glance/images/
total 46076
-rw-r----- 1 root root 13287936 Oct 15 11:59 2e281bf5-daf6-4625-b7c8-edb2e17c2926
-rw-r----- 1 stack stack 4979632 Oct 12 07:43 3ddc5a7e-ce62-4c65-a694-d949b530c08b
-rw-r----- 1 stack stack 25165824 Oct 12 07:43 90235f54-475c-4c56-b1a5-0ffff5ff57161
-rw-r----- 1 stack stack 3740163 Oct 12 07:43 97bd1ddc-bcc9-415f-b42c-e460a6ef0669
```

图 6-16

## 6.4.2 删除 image

### 1. Web UI

admin 登录后，单击 Project → Compute → Images，打开如图 6-17 所示的界面。

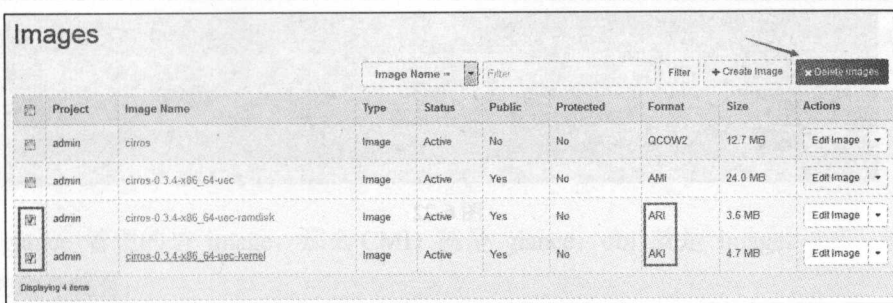


图 6-17

在列表中选择格式为 ARI 和 AKI 的 image，单击“Delete Images”，打开如图 6-18 所示的界面。

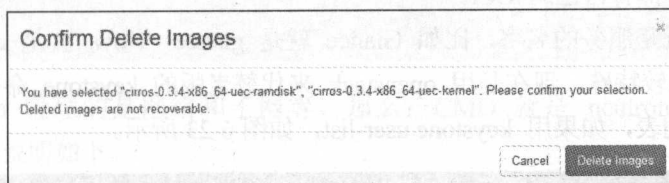


图 6-18

单击“Delete Images”确认删除操作成功，如图 6-19 所示。

Project	Image Name
admin	cirros
admin	cirros-0.3.4-x86_64-uec

Displaying 2 items

图 6-19

## 2. CLI

设置环境变量，如图 6-20 所示。

```
stack@devstack-controller:~$ source devstack/openrc admin admin
```

图 6-20

查询现有 image，如图 6-21 所示。

```
stack@devstack-controller:~$ glance image-list
+-----+-----+
| ID              | Name |
+-----+-----+
| 2e281bf5-daf6-4625-b7c8-edb2e17c2926 | cirros |
| 90235f54-475c-4c56-b1a5-0fff5ff57161 | cirros-0.3.4-x86_64-uec |
+-----+-----+

stack@devstack-controller:~$ ls -l /opt/stack/data/glance/images/
total 37556
-rw-r----- 1 root root 13287936 oct 19 11:59 2e281bf5-daf6-4625-b7c8-edb2e17c2926
-rw-r----- 1 stack stack 25165824 oct 12 07:43 90235f54-475c-4c56-b1a5-0fff5ff57161
```

图 6-21

删除 image，如图 6-22 所示。

```
stack@devstack-controller:~$ glance image-delete 90235f54-475c-4c56-b1a5-0fff5ff57161
stack@devstack-controller:~$ glance image-list
+-----+-----+
| ID | Name |
+-----+-----+
| 2e281bf5-daf6-4625-b7c8-edb2e17c2926 | cirros |
+-----+-----+
stack@devstack-controller:~$ ls -l /opt/stack/data/glance/images/
total 12980
-rw-r----- 1 root:root 13287936 Oct 15 11:59 2e281bf5-daf6-4625-b7c8-edb2e17c2926
```

图 6-22

## 6.5 如何使用 OpenStack CLI

OpenStack 服务都有自己的 CLI。  
命令很好记，就是服务的名字，比如 Glance 就是 `glance`，Nova 就是 `nova`。  
但 Keystone 比较特殊，现在是用 `openstack` 来代替老版的 `keystone` 命令。  
比如查询用户列表，如果用 `keystone user-list`，如图 6-23 所示。

```
stack@devstack-controller:~$ keystone user-list
/usr/local/lib/python2.7/dist-packages/keystoneclient/shell.py:64: DeprecationWarning: The keystone CLI is deprecated in favor of python-openstackclient. For a Python library, continue using python-keystoneclient.
  python-keystoneclient>, DeprecationWarning)
```

图 6-23

会提示 `keystone` 已经 deprecated 了。  
用 `openstack` 命令代替，如图 6-24 所示。

```
stack@devstack-controller:~$ openstack user list
+-----+-----+
| ID | Name |
+-----+-----+
| 46f85f0d946946cfb17817347f592af8 | neutron |
| 4a3c6d8ccdda4c709873378e15b5f73f | demo |
| 855eec3b398848f584ab49bec6a193c0 | admin |
| 89bc61d3f3dc447bb9953e6b83e59b7f | cinder |
| 92db06baa2d54d289ba5d434dff170c2 | glance |
| 9b04b539b3954670bfae413e3087013a | nova |
+-----+-----+
```

图 6-24

不同服务用的命令虽然不同，但这些命令使用方式却非常类似，可以举一反三。

### 1. 执行命令之前，需要设置环境变量。

这些变量包含用户名、Project、密码等。如果不设置，每次执行命令都必须设置相关的命令行参数。

### 2. 各个服务的命令都有增、删、改、查的操作

其格式是：

```
CMD <obj>-create [parm1] [parm2]...
```



```

CMD <obj>-delete [parm]
CMD <obj>-update [parm1] [parm2]...
CMD <obj>-list
CMD <obj>-show [parm]

```

例如 glance 管理的是 image, 那么 CMD 就是 glance; obj 就是 image。  
对应的命令就有:

```

glance image-create
glance image-delete
glance image-update
glance image-list
glance image-show

```

再比如 neutron 管理的是网络和子网等, 那么: CMD 就是 neutron; obj 就是 net 和 subnet。对应的命令说明如下。

网络相关操作:

```

neutron net-create
neutron net -delete
neutron net -update
neutron net -list
neutron net -show

```

子网相关操作:

```

neutron subnet-create
neutron subnet -delete
neutron subnet -update
neutron subnet -list
neutron subnet-show

```

有的命令 <obj> 可以省略, 比如 nova。

下面的操作都是针对 instance:

```

nova boot
nova delete
nova list
nova show

```

### 3. 每个对象都有 ID

delete、show 等操作都以 ID 为参数, 例如图 6-25 所示。

```
stack@devstack-controller:~$ glance image-list
+-----+-----+
| ID | Name |
+-----+-----+
| 2e281bf5-daf6-4625-b7c8-edb2e17c2926 | cirros |
+-----+-----+

stack@devstack-controller:~$ glance image-show 2e281bf5-daf6-4625-b7c8-edb2e17c2926
+-----+-----+
| Property | value |
+-----+-----+
| checksum | ee1eca47dc88f4879d8a229cc70a07c6 |
| container_format | bare |
| created_at | 2015-10-15T03:59:08Z |
| disk_format | qcow2 |
| id | 2e281bf5-daf6-4625-b7c8-edb2e17c2926 |
| min_disk | 0 |
| min_ram | 0 |
| name | cirros |
| owner | 0d1950f32fca4138b67c2b1049b8b3dd |
| protected | False |
| size | 13287936 |
| status | active |
| tags | [] |
| updated_at | 2015-10-15T03:59:08Z |
| virtual_size | none |
| visibility | private |
+-----+-----+
```

图 6-25

4. 可用 help 查看命令的用法

除了 delete、show 等操作只需要 ID 一个参数，其他操作可能需要更多的参数，用 help 查看所需的参数，格式是：

CMD help [SUB-CMD]

例如，查看 glance 都有哪些 SUB-CMD，如图 6-26 所示。

```
stack@devstack-controller:~$ glance help
usage: glance [-version] [-d] [-v] [--get-schema] [--timeout TIMEOUT]
              [--no-ssl-compression] [-f] [--os-image-url OS_IMAGE_URL]
              [--os-image-api-version OS_IMAGE_API_VERSION]
              [--profile HMAC_KEY] [-k] [--os-cert OS_CERT]
              [--cert-file OS_CERT] [--os-key OS_KEY] [--key-file OS_KEY_FILE]
              [--cacert <ca-certificate-file>] [--ca-file OS_CA_FILE]
              [--os-username OS_USERNAME] [--os-user-id OS_USER_ID]
              [--os-user-domain-id OS_USER_DOMAIN_ID]
              [--os-user-domain-name OS_USER_DOMAIN_NAME]
              [--os-project-id OS_PROJECT_ID]
              [--os-project-name OS_PROJECT_NAME]
              [--os-project-domain-id OS_PROJECT_DOMAIN_ID]
              [--os-project-domain-name OS_PROJECT_DOMAIN_NAME]
              [--os-password OS_PASSWORD] [--os-tenant-id OS_TENANT_ID]
              [--os-tenant-name OS_TENANT_NAME] [--os-auth-url OS_AUTH_URL]
              [--os-region-name OS_REGION_NAME]
              [--os-auth-token OS_AUTH_TOKEN]
              [--os-service-type OS_SERVICE_TYPE]
              [--os-endpoint-type OS_ENDPOINT_TYPE]
              <subcommand> ...

Command-line interface to the Openstack Images API.

Positional arguments:
  <subcommand>
    explain          Describe a specific model.
    image-create      Download a specific image.
    image-delete      Delete specified image.
    image-download    Download a specific image.
    image-list        List images you can access.
    image-show        Describe a specific image.
    image-tag-delete  Delete the tag associated with the given image.
    image-tag-update  Update an image with the given tag.
    image-update      Update an existing image.
    image-upload      Upload data for a specific image.
    location-add       Add a location (and related metadata) to an image.
    location-delete   Remove locations (and related metadata) from an image.
    location-update    Update metadata of an image's location.
```

图 6-26

查看 glance image-update 的用法，如图 6-27 所示。

```

stack@devstack-controller:~$ glance help image-update
usage: glance image-update [--architecture <ARCHITECTURE>]
                             [--protected [True|False]] [--name <NAME>]
                             [--instance-uuid <INSTANCE_UUID>]
                             [--min-disk <MIN_DISK>] [--visibility
                             <VISIBILITY>] [--kernel-id <KERNEL_ID>]
                             [--os-version <OS_VERSION>]
                             [--disk-format <DISK_FORMAT>] [--self
                             --os-distro <OS_DISTRO>] [--owner <OWNER>]
                             [--ramdisk-id <RAMDISK_ID>] [--min-ram
                             <MIN_RAM>] [--container-format <CONTAINER_FORMAT>]
                             [--property <key=value>] [--remove-pr
                             <IMAGE_ID>]

Update an existing image.

Positional arguments:
  <IMAGE_ID>          ID of image to update.

Optional arguments:
  --architecture <ARCHITECTURE>
                        Operating system architecture as specified by
                        http://docs.openstack.org/trunk/openstack-compute/admin/content/adding-images.html
  --protected [True|False]
                        If true, image will not be deletable.
  --name <NAME>        Descriptive name for the image
  --instance-uuid <INSTANCE_UUID>
                        ID of instance used to create this image
  --min-disk <MIN_DISK>
                        Amount of disk space (in GB) required to
                        store image
  --visibility <VISIBILITY>
                        Scope of image accessibility valid value
                        private
  --kernel-id <KERNEL_ID>
                        ID of image stored in Glance that should be
                        used as the kernel when booting an AMI-style image
  --os-version <OS_VERSION>
                        Operating system version as specified by
                        distributor
  --disk-format <DISK_FORMAT>
                        Format of the disk valid values: ami, ar
                        vmdk, raw, qcow2, vdi, iso
  --self
                        Update the image that is currently being
                        processed
  --os-distro <OS_DISTRO>
                        Operating system distributor
  --owner <OWNER>      Owner of the image
  --ramdisk-id <RAMDISK_ID>
                        ID of image stored in Glance that should be
                        used as the ramdisk when booting an AMI-style image
  --min-ram <MIN_RAM>
                        Minimum amount of RAM (in GB) required to
                        run the image
  --container-format <CONTAINER_FORMAT>
                        Container format of the image
  --property <key=value>
                        Arbitrary key-value pairs to store with the
                        image
  --remove-pr <IMAGE_ID>
                        Remove the image that is currently being
                        processed

```

图 6-27

## 6.6 如何 Troubleshooting

OpenStack 排查问题的方法主要是通过日志，Service 都有自己单独的日志。Glance 主要有两个日志，glanceapi.log 和 glanceregistry.log，保存在 /var/log/apache2/ 目录里。

devstack 的 screen 窗口已经帮我们打开了这两个日志，可以直接查看，如图 6-28 所示。

```

2015-10-15 16:33:20.676 4314 DEBUG glance
8b3dd - - -] Updating image 3ddc5a7e-ce62
2015-10-15 16:33:20.681 4315 INFO eventlet
-] 192.168.111.22 - - [15/Oct/2015 16:33:
2015-10-15 16:33:20.724 4314 INFO glance
8b3dd - - -] Updating metadata for image 3
2015-10-15 16:33:20.731 4314 INFO eventlet
-] 192.168.111.22 - - [15/Oct/2015 16:33:
2015-10-15 16:33:20.742 4315 DEBUG eventlet
6
2015-10-15 16:33:20.770 4315 INFO glance
8b3dd - - -] Successfully deleted image 3d
2015-10-15 16:33:20.780 4315 INFO eventlet
-] 192.168.111.22 - - [15/Oct/2015 16:33:
2015-10-15 16:33:20.969 4315 DEBUG eventlet
6
2015-10-15 16:33:20.991 4315 DEBUG glance
8b3dd - - -] Returning detailed image lis
2015-10-15 16:33:20.992 4315 INFO eventlet
-] 192.168.111.22 - - [15/Oct/2015 16:33:
45(L) g-reg* 55(L) g-api 65(L) n-api

```

图 6-28

g-api 窗口显示 glance-api 日志，记录 REST API 调用情况。

g-reg 窗口显示 glance-registry 日志，记录 Glance 服务处理请求的过程以及数据库操作。如果需要得到最详细的日志信息，可以在 `/etc/glance/*.conf` 中打开 `debug` 选项。devstack 默认已经打开了 `debug`，如图 6-29 所示。

```
[DEFAULT]  
workers = 2  
registry_host = 192.168.111.22  
rpc_backend = rabbit  
notification_driver = messaging  
image_cache_dir = /opt/stack/data/glance/cache/  
use_syslog = False  
bind_host = 0.0.0.0  
debug = True
```

图 6-29

在非 devstack 安装中，日志在 `/var/log/glance/` 目录里。



# 第 7 章

## ◀ Compute Service—Nova ▶

Compute Service Nova 是 OpenStack 最核心的服务，负责维护和管理云环境的计算资源。

OpenStack 作为 IaaS 的云操作系统，虚拟机生命周期管理也就是通过 Nova 来实现的，如图 7-1 所示。

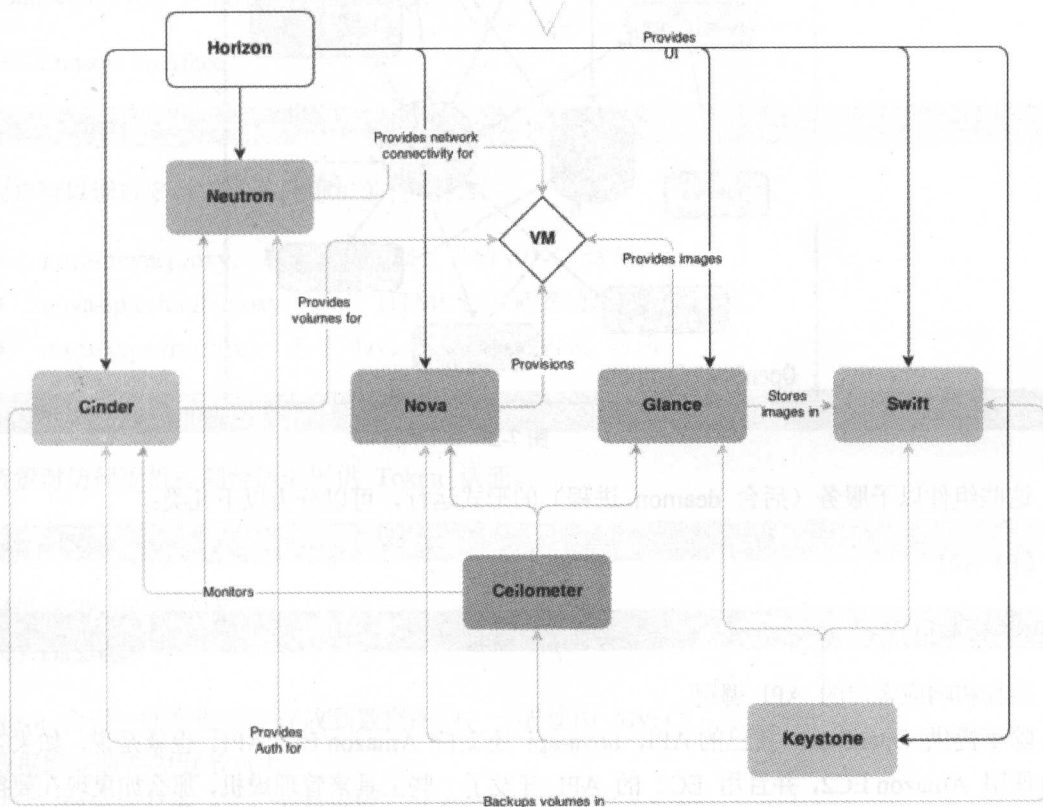


图 7-1

从图 7-1 中可以看到，Nova 处于 OpenStack 架构的中心，其他组件都为 Nova 提供支持：

- Glance 为 VM 提供 image。
- Cinder 和 Swift 分别为 VM 提供块存储和对象存储。
- Neutron 为 VM 提供网络连接。

# 7.1 Nova 架构

本节从架构层次讨论 Nova 服务。

## 7.1.1 架构概览

Nova 的架构比较复杂，包含很多组件，如图 7-2 所示。

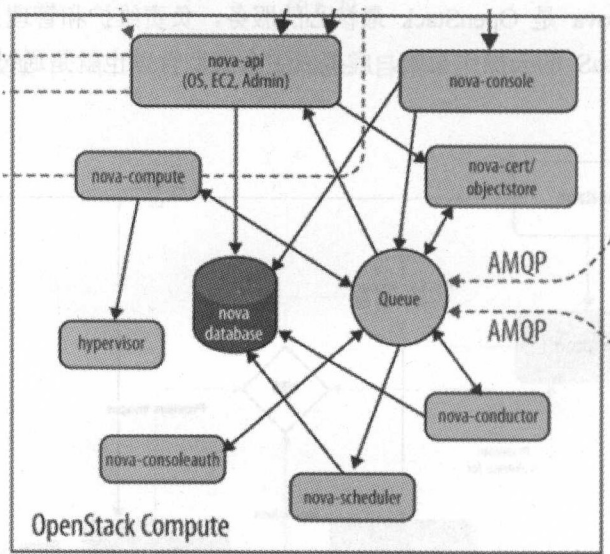


图 7-2

这些组件以子服务（后台 daemon 进程）的形式运行，可以分为以下几类：

### (1) API

`nova-api`

接收和响应客户的 API 调用。

除了提供 OpenStack 自己的 API，nova-api 还支持 Amazon EC2 API。也就是说，如果客户以前使用 Amazon EC2，并且用 EC2 的 API 开发了一些工具来管理虚拟机，那么如果现在要换成 OpenStack，这些工具可以无缝地迁移到 OpenStack，因为 nova-api 兼容 EC2 API，无须做任何修改。

### (2) Compute Core

`nova-scheduler`

虚拟机调度服务，负责决定在哪个计算节点上运行虚拟机。

## nova-compute

管理虚机的核心服务，通过调用 Hypervisor API 实现虚机生命周期管理。

## Hypervisor

计算节点上跑的虚拟化管理程序，虚机管理最底层的程序。

不同虚拟化技术提供自己的 Hypervisor。常用的 Hypervisor 有 KVM、Xen、VMWare 等。

## nova-conductor

nova-compute 经常需要更新数据库，比如更新虚机的状态。

出于安全性和伸缩性的考虑，nova-compute 并不会直接访问数据库，而是将这个任务委托给 nova-conductor，这个我们在后面会详细讨论。

### (3) Console Interface

## nova-console

用户可以通过多种方式访问虚机的控制台：

- nova-novncproxy: 基于 Web 浏览器的 VNC 访问。
- nova-spicehtml5proxy: 基于 HTML5 浏览器的 SPICE 访问。
- nova-xvpngvncproxy: 基于 Java 客户端的 VNC 访问。

## nova-consoleauth

负责对访问虚机控制台请求提供 Token 认证。

## nova-cert

提供 x509 证书支持。

### (4) Database

Nova 会有一些数据需要存放到数据库中，一般使用 MySQL。

数据库安装在控制节点上。

Nova 使用命名为 nova 的数据库，其表的信息如图 7-3 所示。

```

root@devstack-controller:~# su - stack
stack@devstack-controller:~$ mysql
Welcome to the MySQL monitor.  Commands end with ; or
your MySQL connection id is 40
Server version: 5.5.44-0ubuntu0.14.04.1 (Ubuntu)

Copyright (c) 2000, 2015, Oracle and/or its affiliates
Oracle is a registered trademark of Oracle Corporation
and/or its affiliates. Other names may be trademarks of their
respective owners.

Type 'help;' or '\h' for help. Type '\c' to clear the
current input statement.
mysql> use nova
Database changed
mysql> show tables;
+-----+
Tables_in_nova
+-----+
agent_builds
aggregate_hosts
aggregate_metadata
aggregates
block_device_mapping
bw_usage_cache
cells
certificates
compute_nodes
console_pools
consoles
dns_domains
fixed_ips
floating_ips
instance_actions
instance_actions_events
+-----+

```

图 7-3

### (5) Message Queue

在前面我们了解到 Nova 包含众多的子服务，这些子服务之间需要相互协调和通信。为解耦各个子服务，Nova 通过 Message Queue 作为子服务的信息中转站。

所以在架构图上我们看到了子服务之间没有直接的连线，它们都通过 Message Queue 联系，如图 7-4 所示。

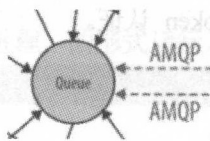


图 7-4

OpenStack 默认是用 RabbitMQ 作为 Message Queue。

Message Queue 是 OpenStack 的核心基础组件，我们后面也会详细介绍。

## 7.1.2 物理部署方案

前面大家已经看到 Nova 由很多子服务组成，同时我们也知道 OpenStack 是一个分布式系统，可以部署到若干节点上，那么接下来大家可能就会问：Nova 的这些服务在物理上应该如何部署呢？

对于 Nova，这些服务会部署在两类节点上：计算节点和控制节点。

计算节点上安装了 Hypervisor，上面运行虚拟机。

由此可知：



- (1) 只有 nova-compute 需要放在计算节点上。
- (2) 其他子服务则是放在控制节点上的。

下面我们可以看看实验环境的具体部署情况。

通过在计算节点和控制节点上运行 `ps -e|grep nova` 命令来查看运行的 nova 子服务。

- 计算节点上命令执行结果如图 7-5 所示。

```
root@devstack-compute1:~# ps -e|grep nova
2970 pts/5    00:24:55 nova-compute
root@devstack-compute1:~#
```

图 7-5

计算节点 devstack-compute1 上只运行了 nova-compute 子服务。

- 控制节点上命令执行结果如图 7-6 所示。

```
root@devstack-controller:~# ps -e|grep nova
17123 pts/20    00:00:00 nova-novncproxy
17128 pts/18    00:00:02 nova-cert
17129 pts/17    00:00:02 nova-conductor
17132 pts/10    00:00:02 nova-api
17134 pts/19    00:00:02 nova-scheduler
17137 pts/21    00:00:02 nova-consoleaut
17146 pts/22    00:00:02 nova-compute
```

图 7-6

控制节点 devstack-controller 上运行了若干 nova-\* 子服务，如图 7-7 所示。

```
root@devstack-controller:~# ps -e|grep rabbit
3055 ?          00:00:00 rabbitmq-server
root@devstack-controller:~# ps -e|grep mysql
2297 ?          00:00:37 mysqld
root@devstack-controller:~#
```

图 7-7

RabbitMQ 和 MySQL 也是放在控制节点上的。

可能细心的同学已经从控制节点上命令执行结果中发现我们的控制节点上也运行了 nova-compute。

这实际上也就意味着 devstack-controller 既是一个控制节点，也是一个计算节点，也可以在上面运行虚拟机。

这也向我们展示了 OpenStack 这种分布式架构部署上的灵活性：

- 可以将所有服务都放在一台物理机上，作为一个 All-in-One 的测试环境；
- 也可以将服务部署在多台物理机上，获得更好的性能和高可用。

另外，也可以用 `nova service-list` 查看 nova-\* 子服务都分布在哪些节点上，如图 7-8 所示。

```
root@devstack-controller:~# source /opt/stack/devstack/openrc admin admin
root@devstack-controller:~# nova service-list
```

Id	Binary	Host	Zone	Status	State
1	nova-conductor	devstack-controller	internal	enabled	up
2	nova-cert	devstack-controller	internal	enabled	up
3	nova-scheduler	devstack-controller	internal	enabled	up
4	nova-consoleauth	devstack-controller	internal	enabled	up
5	nova-compute	devstack-controller	nova	enabled	up
6	nova-compute	devstack-compute1	nova	enabled	up

```
root@devstack-controller:~#
```

图 7-8

7.1.3 从虚拟机创建流程看 nova-\* 子服务如何协同工作

从学习 Nova 的角度看，虚拟机创建是一个非常好的场景，涉及的 nova-\* 子服务很全，如图 7-9 所示流程图。

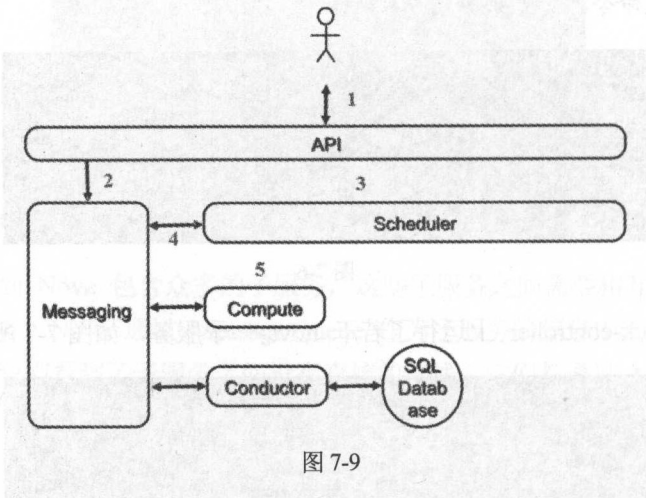


图 7-9

- 客户（可以是 OpenStack 最终用户，也可以是其他程序）向 API (nova-api) 发送请求：“帮我创建一个虚拟机”。
- API 对请求做一些必要处理后，向 Messaging (RabbitMQ) 发送了一条消息：“让 Scheduler 创建一个虚拟机”。
- Scheduler (nova-scheduler) 从 Messaging 获取到 API 发给它的消息，然后执行调度算法，从若干计算节点中选出节点 A。
- Scheduler 向 Messaging 发送了一条消息：“在计算节点 A 上创建这个虚拟机”。
- 计算节点 A 的 Compute (nova-compute) 从 Messaging 中获取到 Scheduler 发给它的消息，然后在本节点的 Hypervisor 上启动虚拟机。
- 在虚拟机创建的过程中，Compute 如果需要查询或更新数据库信息，会通过 Messaging 向 Conductor (nova-conductor) 发送消息，Conductor 负责数据库访问。

上面是创建虚拟机最核心的几个步骤，当然也省略了很多细节，我们会在后面的章节详细讨论。这几个步骤向我们展示了 nova-\* 子服务之间的协作的方式，也体现了 OpenStack 整个系

统的分布式设计思路，掌握这种思路对我们深入理解 OpenStack 会非常有帮助。

## 7.1.4 OpenStack 通用设计思路

### 1. API 前端服务

每个 OpenStack 组件可能包含若干子服务，其中必定有一个 API 服务负责接收客户请求。

以 Nova 为例，nova-api 作为 Nova 组件对外的唯一窗口，向客户暴露 Nova 能够提供的功能。

当客户需要执行虚机相关的操作，能且只能向 nova-api 发送 REST 请求。

这里的客户包括终端用户、命令行和 OpenStack 其他组件。

设计 API 前端服务的好处在于：

- (1) 对外提供统一接口，隐藏实现细节。
- (2) API 提供 REST 标准调用服务，便于与第三方系统集成。
- (3) 可以通过运行多个 API 服务实例轻松实现 API 的高可用，比如运行多个 nova-api 进程。

### 2. Scheduler 调度服务

对于某项操作，如果有多个实体都能够完成任务，那么通常会有一个 scheduler 负责从这些实体中挑选出一个最合适的来执行操作。

在前面的例子中，Nova 有多个计算节点。当需要创建虚机时，nova-scheduler 会根据计算节点当时的资源使用情况选择一个最合适的计算节点来运行虚机。

调度服务就好比是一个开发团队中的项目经理，当接到新的开发任务时，项目经理会评估任务的难度，考察团队成员目前的工作负荷和技能水平，然后将任务分配给最合适的开发人员。

除了 Nova，块服务组件 Cinder 也有 scheduler 子服务，后面我们会详细讨论。

### 3. Worker 工作服务

调度服务只管分配任务，真正执行任务的是 Worker 工作服务。

在 Nova 中，这个 Worker 就是 nova-compute 了。

将 Scheduler 和 Worker 从职能上进行划分使得 OpenStack 非常容易扩展：

- 当计算资源不够了无法创建虚机时，可以增加计算节点（增加 Worker）
- 当客户的请求量太大调度不过来时，可以增加 Scheduler

### 4. Driver 框架

OpenStack 作为开放的 Infrastructure as a Service 云操作系统，支持业界各种优秀的技术。

这些技术可能是开源免费的，也可能是商业收费的。

这种开放的架构使得 OpenStack 能够在技术上保持先进性，具有很强的竞争力，同时又不会造成厂商锁定（Lock-in）。

那么 OpenStack 的这种开放性体现在哪里呢？一个重要的方面就是采用基于 Driver 的框架。

以 Nova 为例，OpenStack 的计算节点支持多种 Hypervisor。包括 KVM、Hyper-V、VMWare、Xen、Docker、LXC 等。

nova-compute 为这些 Hypervisor 定义了统一的接口，hypervisor 只需要实现这些接口，就可以 driver 的形式即插即用 OpenStack 中。

如图 7-10 所示是 nova driver 的架构示意图。

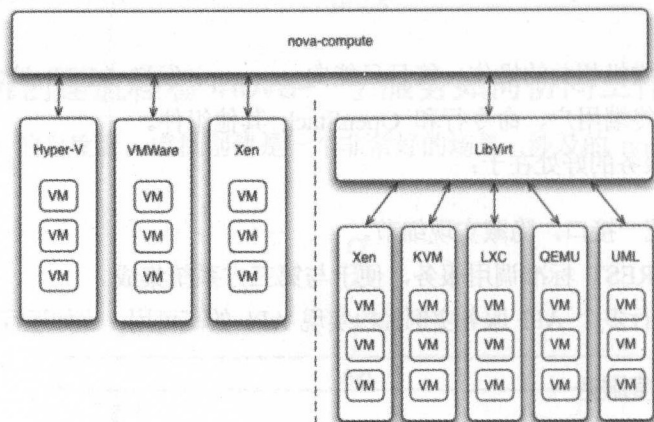


图 7-10

在 nova-compute 的配置文件 `/etc/nova/nova.conf` 中，由 `compute_driver` 配置项指定该计算节点使用哪种 Hypervisor 的 driver，如图 7-11 所示。

```
compute_driver = libvirt.LibvirtDriver
```

图 7-11

在我们的环境中因为是 KVM，所以配置的是 Libvirt 的 driver。

不知大家是否还记得我们在学习 Glance 时谈到：OpenStack 支持多种 backend 来存放 image，可以是本地文件系统、Cinder、Ceph、Swift 等。其实这这也是一个 driver 架构，只要符合 Glance 定义规范，新的存储方式可以很方便地加入到 backend 支持列表中。

在后面 Cinder 和 Neutron 中我们还会看到 driver 框架的应用。

## 5. Messaging 服务

在前面创建虚机的流程示意图中，我们看到 nova-\* 子服务之间的调用严重依赖 Messaging。Messaging 是 nova-\* 子服务交互的中枢，如图 7-12 所示。



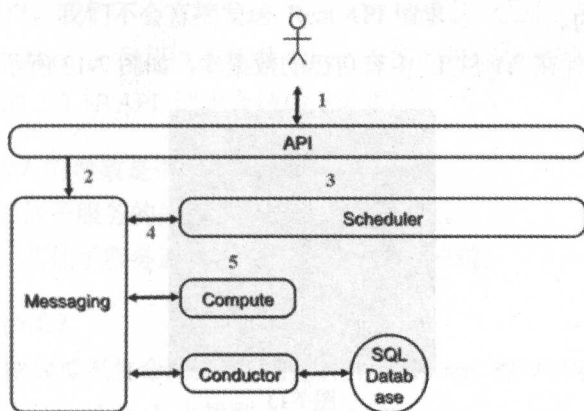


图 7-12

以前没接触过分布式系统的同学可能会不太理解，为什么不让 API 直接调用 Scheduler，或是让 Scheduler 直接调用 Compute，而是非要通过 Messaging 进行中转，这里做一些解释。

程序之间的调用通常分两种：同步调用和异步调用。

#### ● 同步调用

API 直接调用 Scheduler 的接口就是同步调用。其特点是 API 发出请求后需要一直等待，直到 Scheduler 完成对 Compute 的调度，将结果返回给 API 后 API 才能够继续做后面的工作。

#### ● 异步调用

API 通过 Messaging 间接调用 Scheduler 就是异步调用。其特点是 API 发出请求后不需要等待，直接返回，继续做后面的工作。

Scheduler 从 Messaging 接收到请求后执行调度操作，完成后将结果也通过 Messaging 发送给 API。

在 OpenStack 这类分布式系统中，通常采用异步调用的方式，其好处是：

##### (1) 解耦各子服务

子服务不需要知道其他服务在哪里运行，只需要发送消息给 Messaging 就能完成调用。

##### (2) 提高性能

异步调用使得调用者无须等待结果返回。这样可以继续执行更多的工作，提高系统总的吞吐量。

##### (3) 提高伸缩性

子服务可以根据需要进行扩展，启动更多的实例处理更多的请求，在提高可用性的同时也提高了整个系统的伸缩性。而且这种变化不会影响到其他子服务，也就是说变化对别人是透明的。

在后面各章节，我们都能看到 Messaging 的应用。

## 6. Database

OpenStack 各组件都需要维护自己的状态信息。比如 Nova 中有虚机的规格、状态，这些信

息都是在数据库中维护的。

每个 OpenStack 组件在 MySQL 中有自己的数据库，如图 7-13 所示。

```
mysql> show databases;
+-----+
Database
+-----+
information_schema
cinder
glance
keystone
mysql
neutron
nova
nova_api
performance_schema
+-----+
```

图 7-13

7. 小结

Nova 是 OpenStack 中最重要的组件，也是很典型的组件。Nova 充分体现了 OpenStack 的设计思路。理解了这种思路，再来学习 OpenStack 的其他组件就能够举一反三，清晰容易很多。我们在后面 Cinder 和 Neutron 的学习中还会回顾这些设计思路。

# 7.2 Nova 组件详解

本节开始，我们将详细讲解 Nova 的各个子服务。

从前面架构概览一节，我们知道 Nova 有若干 nova-\* 的子服务，下面我们将依次学习最重要的几个子服务。

## 7.2.1 nova-api

nova-api 是整个 Nova 组件的门户，所有对 Nova 的请求都首先由 nova-api 处理。nova-api 向外界暴露若干 HTTP REST API 接口。

在 keystone 中我们可以查询 nova-api 的 endpoints，如图 7-14 所示。

```
root@devstack-controller:/opt/stack/devstack# source openrc admin admin
root@devstack-controller:/opt/stack/devstack# openstack endpoint show nova
+-----+-----+
Field      Value
+-----+-----+
adminurl   http://192.168.111.22:8774/v2.1/${tenant_id}s
enabled    True
id         2491e7c434754282b99ea332d424aaae
internalurl http://192.168.111.22:8774/v2.1/${tenant_id}s
publicurl  http://192.168.111.22:8774/v2.1/${tenant_id}s
region     RegionOne
service_id a254b2be2bf2423d8bcbaaa9b5bd2538
service_name nova
service_type compute
+-----+-----+
```

图 7-14

客户端就可以将请求发送到 endpoints 指定的地址，向 nova-api 请求操作。

当然，作为最终用户，我们不会直接发送 Rest API 请求。

OpenStack CLI、Dashboard 和其他需要跟 Nova 交换的组件会使用这些 API。

nova-api 对接收到的 HTTP API 请求会做如下处理：

- (1) 检查客户端传入的参数是否合法有效。
- (2) 调用 Nova 其他子服务的处理客户端 HTTP 请求。
- (3) 格式化 Nova 其他子服务返回的结果并返回给客户端。

nova-api 接收哪些请求？

简单地说，只要是跟虚拟机生命周期相关的操作，nova-api 都可以响应。

大部分操作都可以在 Dashboard 上找到。

打开 Instance 管理界面，如图 7-15 所示。

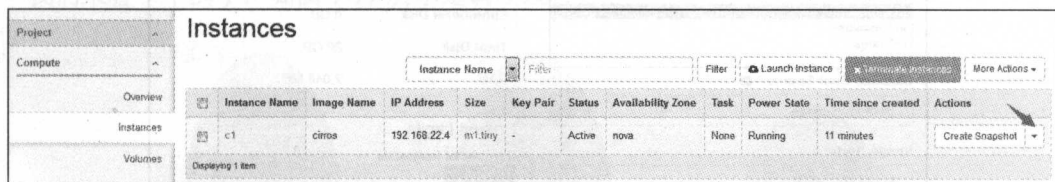


图 7-15

单击下拉箭头，列表中就是 nova-api 可执行的操作，如图 7-16 所示。

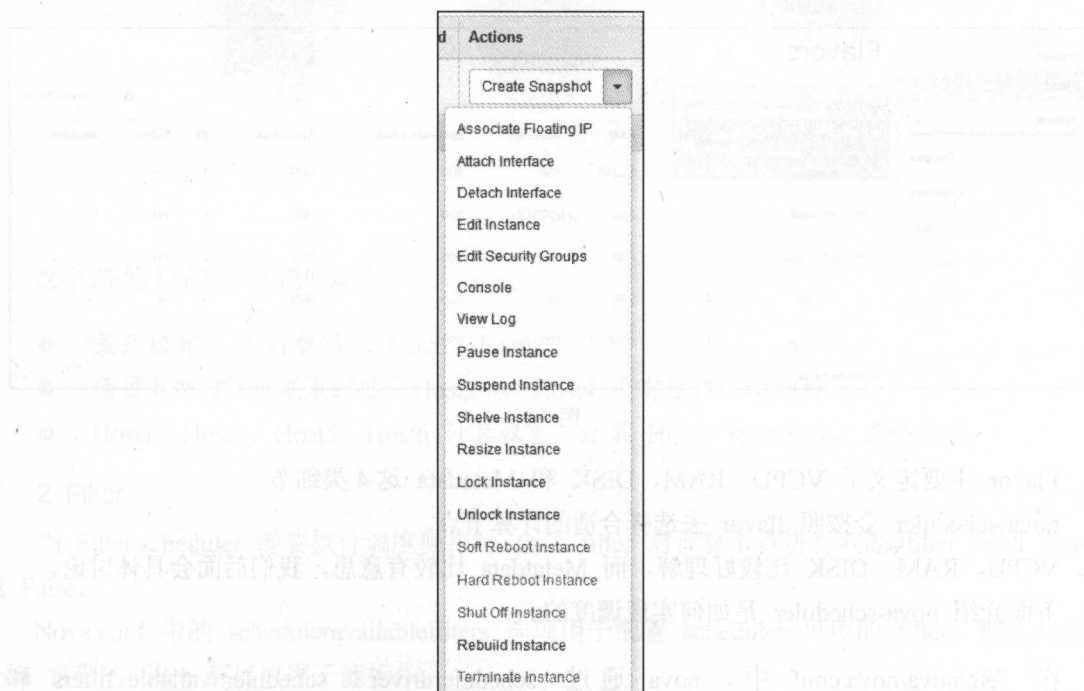


图 7-16

OpenStack 用术语 Instance 来表示虚拟机，后面我们将统一使用这个术语。

7.2.2 nova-scheduler

创建 Instance 时，用户会提出资源需求，例如 CPU、内存、磁盘各需要多少。OpenStack 将这些需求定义在 flavor 中，用户只需要指定用哪个 flavor 就可以了，如图 7-17 所示。

Launch Instance

Details \*

Access & Security

Networking \*

Post-Creation

Advanced Options

Availability Zone

Any Availability Zone

Specify the details for launching an instance.

The chart below shows the resources used by this project in relation to the project's quotas.

Flavor Details

Name

m1.small

VCPUs

1

Root Disk

20 GB

Ephemeral Disk

0 GB

Total Disk

20 GB

RAM

2,048 MB

Instance Name \*

c1

Flavor \*

m1.small

m1.tiny

m1.small

m1.medium

m1.large

m1.xlarge

Instance Boot Source \*

Boot from image

Project Limits

Number of Instances

1 of 10 Used

Image Name \*

cirros (12.7 MB)

图 7-17

可用的 flavor 在 System → Flavors 中管理，如图 7-18 所示。

Project

Admin

System

Overview

Hypervisors

Host Aggregates

Instances

Volumes

Flavors

Images

Flavors

Filter

Q

+ Create Flavor

Flavor Name	VCPUs	RAM	Root Disk	Ephemeral Disk	Swap Disk	ID	Public	Metadata
m1.tiny	1	512MB	1GB	0GB	0MB	1	Yes	Yes
m1.small	1	2GB	20GB	0GB	0MB	2	Yes	No
m1.medium	2	4GB	40GB	0GB	0MB	3	Yes	No
m1.large	4	8GB	80GB	0GB	0MB	4	Yes	No
m1.xlarge	8	16GB	160GB	0GB	0MB	5	Yes	No

Displaying 5 items

图 7-18

Flavor 主要定义了 VCPU、RAM、DISK 和 Metadata 这 4 类细节。nova-scheduler 会按照 flavor 去选择合适的计算节点。VCPU、RAM、DISK 比较好理解，而 Metatdata 比较有意思，我们后面会具体讨论。下面介绍 nova-scheduler 是如何实现调度的。

在 /etc/nova/nova.conf 中，nova 通过 schedulerdriver、scheduleravailable\_filters 和 schedulerdefaultfilters 这三个参数来配置 nova-scheduler。



## 1. Filter scheduler

Filter scheduler 是 nova-scheduler 默认的调度器，调度过程分为两步：

- 通过过滤器 (filter) 选择满足条件的计算节点 (运行 nova-compute)
- 通过权重计算 (weighting) 选择在最优 (权重值最大) 的计算节点上创建 Instance。

```
scheduler_driver=nova.scheduler.filterscheduler.FilterScheduler
```

Nova 允许使用第三方 scheduler，配置 scheduler\_driver 即可。

这又一次体现了 OpenStack 的开放性。

Scheduler 可以使用多个 filter 依次进行过滤，过滤之后的节点再通过计算权重选出最适合的节点。

我们来看一个例子，如图 7-19 所示是调度过程。

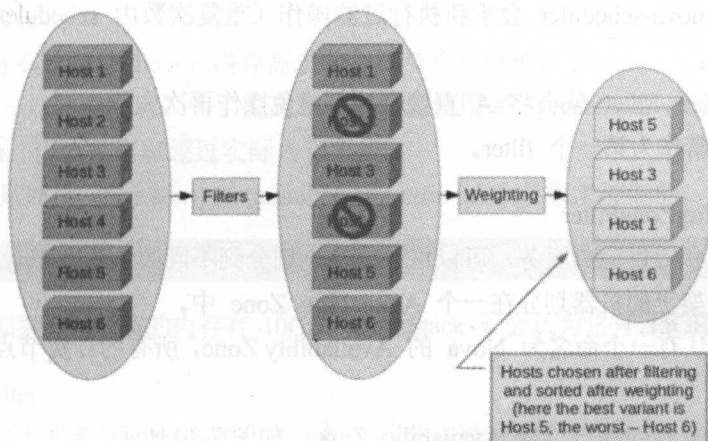


图 7-19

这个调度过程的示例说明如下：

- 最开始有 6 个计算节点 Host1~Host6。
- 通过多个 filter 层层过滤，Host2 和 Host4 没有通过，被刷掉了。
- Host1、Host3、Host5、Host6 计算权重，结果 Host5 得分最高，最终入选。

## 2. Filter

当 Filter scheduler 需要执行调度操作时，会让 filter 对计算节点进行判断，filter 返回 True 或 False。

Nova.conf 中的 scheduleravailablefilters 选项用于配置 scheduler 可用的 filter，默认是所有 nova 自带的 filter 都可以用于滤操作。

```
scheduler_available_filters = nova.scheduler.filters.all_filters
```

另外还有一个选项 schedulerdefaultfilters 用于指定 scheduler 真正使用的 filter，默认值如下：

```
scheduler_default_filters = RetryFilter, AvailabilityZoneFilter,
RamFilter, DiskFilter, ComputeFilter, ComputeCapabilitiesFilter,
ImagePropertiesFilter, ServerGroupAntiAffinityFilter,
ServerGroupAffinityFilter
```

Filter scheduler 将按照列表中的顺序依次过滤。

下面依次介绍每个 filter。

(1) RetryFilter

RetryFilter 的作用是刷掉之前已经调度过的节点。

举个例子方便大家理解：

假设 A、B、C 三个节点都通过了过滤，最终 A 因为权重值最大被选中执行操作。

但由于某个原因，操作在 A 上失败了。

默认情况下，nova-scheduler 会重新执行过滤操作（重复次数由 schedulermaxattempts 选项指定，默认是 3）。

那么这时候 RetryFilter 就会将 A 直接刷掉，避免操作再次失败。

RetryFilter 通常作为第一个 filter。

(2) AvailabilityZoneFilter

为提高容灾性和提供隔离服务，可以将计算节点划分到不同的 Availability Zone 中。

例如把一个机架上的机器划分在一个 Availability Zone 中。

OpenStack 默认有一个命名为 Nova 的 Availability Zone，所有的计算节点初始都放在 Nova 中。

用户可以根据需要创建自己的 Availability Zone，如图 7-20 所示。

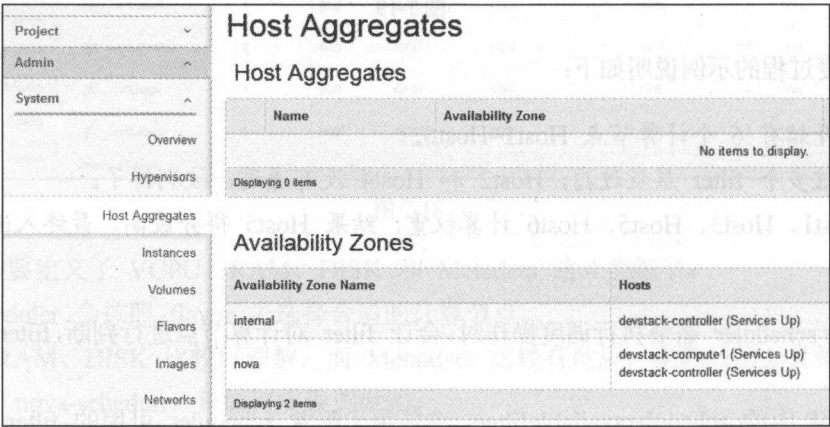


图 7-20

创建 Instance 时，需要指定将 Instance 部署到在哪个 Availability Zone 中，如图 7-21 所示。

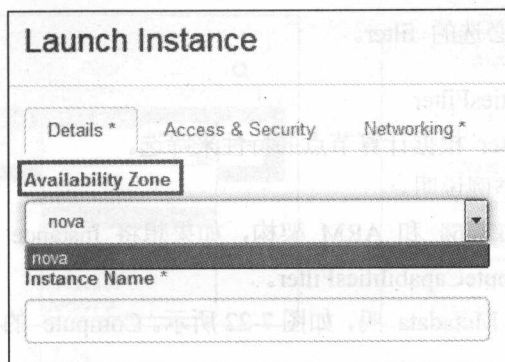


图 7-21

nova-scheduler 在做 filtering 时, 会使用 AvailabilityZoneFilter 将不属于指定 Availability Zone 的计算节点过滤掉。

### (3) RamFilter

RamFilter 将不能满足 flavor 内存需求的计算节点过滤掉。

对于内存有一点需要注意: 为了提高系统的资源使用率, OpenStack 在计算节点可用内存时允许 overcommit, 也就是可以超过实际内存大小。

超过的程度是通过 nova.conf 中 ramallocationratio 这个参数来控制的, 默认值为 1.5。

```
ram_allocation_ratio = 1.5
```

其含义是: 如果计算节点的内存有 10GB, OpenStack 则会认为它有 15GB (10\*1.5) 的内存。

### (4) DiskFilter

DiskFilter 将不能满足 flavor 磁盘需求的计算节点过滤掉。

Disk 同样允许 overcommit, 通过 nova.conf 中 diskallocationratio 控制, 默认值为 1。

```
Disk_allocation_ratio = 1.0
```

### (5) CoreFilter

CoreFilter 将不能满足 flavor vCPU 需求的计算节点过滤掉。

vCPU 同样允许 overcommit, 通过 nova.conf 中 cpuallocationratio 控制, 默认值为 16。

```
Cpu_allocation_ratio = 16.0
```

这意味着一个 8 vCPU 的计算节点, nova-scheduler 在调度时认为它有 128 个 vCPU。

需要提醒的是: nova-scheduler 默认使用的 filter 并没有包含 CoreFilter。如果要用, 可以将 CoreFilter 添加到 nova.conf 的 schedulerdefaultfilters 配置选项中。

### (6) ComputeFilter

ComputeFilter 保证只有 nova-compute 服务正常工作的计算节点, 才能够被 nova-scheduler 调度。

ComputeFilter 显然是必选的 filter。

(7) ComputeCapabilitiesFilter

ComputeCapabilitiesFilter 根据计算节点的特性来筛选。

这个比较高级，我们举例说明。

例如，我们的节点有 x8664 和 ARM 架构，如果想将 Instance 指定部署到 x8664 架构的节点上，就可以利用 ComputeCapabilitiesFilter。

还记得 flavor 中有个 Metadata 吗，如图 7-22 所示。Compute 的 Capabilities 就在 Metadata 中指定。

Compute Host Capabilities 列出了所有可设置 Capabilities，如图 7-23 所示。

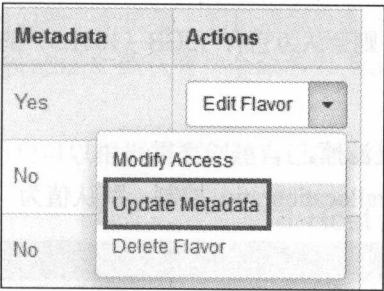


图 7-22

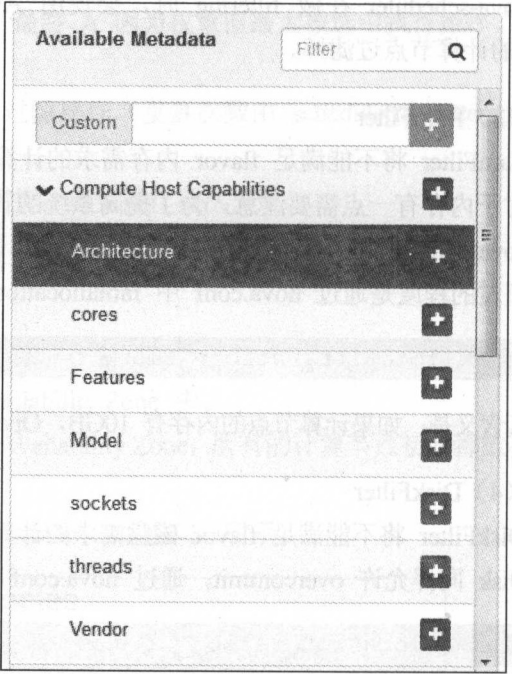


图 7-23

单击 Architecture 后面的“+”，就可以在右边的列表中指定具体的架构，如图 7-24 所示。配置好后，ComputeCapabilitiesFilter 在调度时只会筛选出 x86\_64 的节点。如果没有设置 Metadata，ComputeCapabilitiesFilter 不会起作用，所有节点都会通过筛选。

(8) ImagePropertiesFilter

ImagePropertiesFilter 根据所选 image 的属性来筛选匹配的计算节点。

跟 flavor 类似，image 也有 metadata，用于指定其属性，如图 7-25 所示。



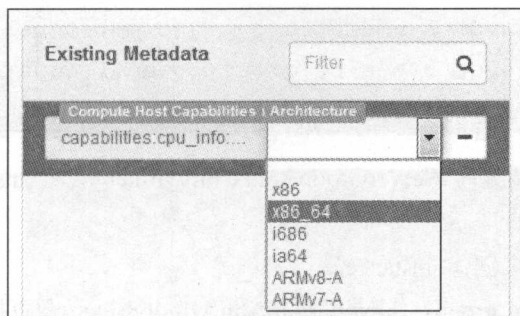


图 7-24

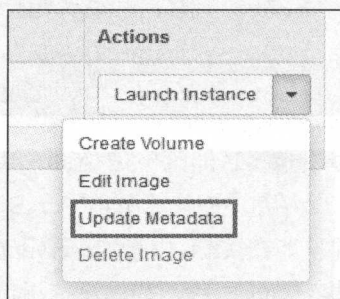


图 7-25

例如，希望某个 image 只能运行在 kvm 的 hypervisor 上，可以通过 Hypervisor Type 属性来指定，如图 7-26 所示。

单击“+”，然后在右边的列表中选择 kvm，如图 7-27 所示。

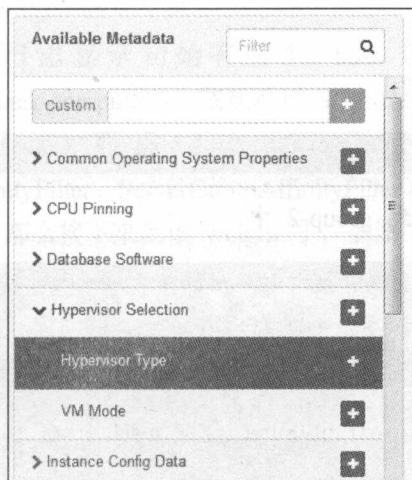


图 7-26

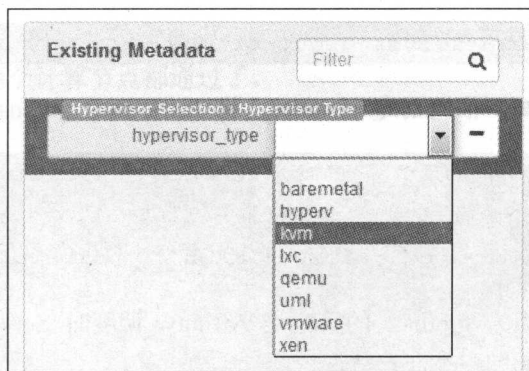


图 7-27

配置好后，ImagePropertiesFilter 在调度时只会筛选出 kvm 的节点。

如果没有设置 Image 的 Metadata，ImagePropertiesFilter 不会起作用，所有节点都会通过筛选。

#### (9) ServerGroupAntiAffinityFilter

ServerGroupAntiAffinityFilter 可以尽量将 Instance 分散部署到不同的节点上。

例如有 inst1、inst2 和 inst3 三个 instance，计算节点有 A、B 和 C。

为保证分散部署，进行如下操作：

- 创建一个 anti-affinity 策略的 server group “group-1”。

```
nova server-group-create --policy anti-affinity group-1
```

请注意，这里的 server group 其实是 instance group，并不是计算节点的 group。

- 依次创建 Instance，将 inst1、inst2 和 inst3 放到 group-1 中。

```
nova boot --image IMAGEID --flavor 1 --hint group=group-1 inst1
nova boot --image IMAGEID --flavor 1 --hint group=group-1 inst2
nova boot --image IMAGE_ID --flavor 1 --hint group=group-1 inst3
```

因为 group-1 的策略是 AntiAffinity, 调度时 ServerGroupAntiAffinityFilter 会将 inst1、inst2 和 inst3 部署到不同计算节点 A、B 和 C。

目前只能在 CLI 中指定 server group 来创建 instance。

创建 instance 时, 如果没有指定 server group, ServerGroupAntiAffinityFilter 会直接通过, 不做任何过滤。

#### (10) ServerGroupAffinityFilter

与 ServerGroupAntiAffinityFilter 的作用相反, ServerGroupAffinityFilter 会尽量将 instance 部署到同一个计算节点上。

方法类似这个:

- 创建一个 affinity 策略的 server group“group-2”。

```
nova server-group-create --policy affinity group-2
```

- 依次创建 instance, 将 inst1、inst2 和 inst3 放到 group-2 中。

```
nova boot --image IMAGEID --flavor 1 --hint group=group-2 inst1
nova boot --image IMAGEID --flavor 1 --hint group=group-2 inst2
nova boot --image IMAGE_ID --flavor 1 --hint group=group-2 inst3
```

因为 group-2 的策略是 Affinity, 调度时 ServerGroupAffinityFilter 会将 inst1、inst2 和 inst3 部署到同一个计算节点。

创建 instance 时如果没有指定 server group, ServerGroupAffinityFilter 会直接通过, 不做任何过滤。

### 3. Weight

经过前面一堆 filter 的过滤, nova-scheduler 选出了能够部署 instance 的计算节点。

如果有多个计算节点通过了过滤, 那么最终选择哪个节点呢?

Scheduler 会对每个计算节点打分, 得分最高的获胜。

打分的过程就是 weight, 翻译过来就是计算权重值, 那么 scheduler 是根据什么来计算权重值呢?

目前 nova-scheduler 的默认实现是根据计算节点空闲的内存量计算权重值:

空闲内存越多, 权重越大, instance 将被部署到当前空闲内存最多的计算节点上。

### 4. 日志

是时候完整地回顾一下 nova-scheduler 的工作过程了。

整个过程都被记录到 nova-scheduler 的日志中。

比如,当我们部署一个 instance 时,打开 nova-scheduler 的日志 /opt/stack/logs/n-sch.log (非 devstack 安装其日志在 /var/log/nova/scheduler.log), 如图 7-28 所示。

```
admin admin] Starting with 2 host(s) get_filtered_objects /opt/stack/nova/nova/fi
admin admin] Filter RetryFilter returned 2 host(s) get_filtered_objects /opt/stack
admin admin] Filter AvailabilityZoneFilter returned 2 host(s) get_filtered_object
admin admin] Filter RamFilter returned 2 host(s) get_filtered_objects /opt/stack/
admin admin] Filter DiskFilter returned 2 host(s) get_filtered_objects /opt/stack
admin admin] Filter ComputeFilter returned 2 host(s) get_filtered_objects /opt/st
admin admin] Filter ComputeCapabilitiesFilter returned 2 host(s) get_filtered_obj
admin admin] Filter ImagePropertiesFilter returned 2 host(s) get_filtered_objects
admin admin] Filter ServerGroupAntiAffinityFilter returned 2 host(s) get_filtered
admin admin] Filter ServerGroupAffinityFilter returned 2 host(s) get_filtered_obj
```

图 7-28

日志显示初始有两个 host (在我们的实验环境中就是 devstack-controller 和 devstack-compute1), 依次经过 9 个 filter 的过滤 (RetryFilter、AvailabilityZoneFilter、RamFilter、DiskFilter、ComputeFilter、ComputeCapabilitiesFilter、ImagePropertiesFilter、ServerGroupAntiAffinityFilter、ServerGroupAffinityFilter), 两个计算节点都通过了。

那么接下来就该 weight 了, 如图 7-29 所示。

```
2015-12-10 15:51:16.957 DEBUG nova.scheduler.filter_scheduler [req-4915ac4f-6a25-
4c86-ad0f-b116001812e6 admin admin] weighed [weighedHost [host: (devstack-control
ler, devstack-controller) ram:7466 disk:88064 io_ops:0 instances:0, weight: 1.0],
weighedHost [host: (devstack-compute1, devstack-compute1) ram:3434 disk:90112 io
_ops:0 instances:0, weight: 0.459951781499]] _schedule /opt/stack/nova/nova/sched
uler/filter_scheduler.py:157
2015-12-10 15:51:16.959 DEBUG nova.scheduler.filter_scheduler [req-4915ac4f-6a25-
4c86-ad0f-b116001812e6 admin admin] Selected host: weighedHost [host: (devstack-c
ontroller, devstack-controller) ram:7466 disk:88064 io_ops:0 instances:0, weight:
1.0] _schedule /opt/stack/nova/nova/scheduler/filter_scheduler.py:167
```

图 7-29

可以看到因为 devstack-controller 的空闲内存比 devstack-compute1 多 ( $7466 > 3434$ ), 权重值更大 ( $1.0 > 0.4599$ ), 最终选择 devstack-controller。

注意: 要显示 DEBUG 日志, 需要在 /etc/nova/nova.conf 中打开 debug 选项。

```
[DEFAULT]
debug = True
```

## 7.2.3 nova-compute

nova-compute 在计算节点上运行, 负责管理节点上的 instance。

OpenStack 对 instance 的操作, 最后都是交给 nova-compute 来完成的。

nova-compute 与 Hypervisor 一起实现 OpenStack 对 instance 生命周期的管理。

1. 通过 Driver 架构支持多种 Hypervisor

接着的问题是：现在市面上有这么多 Hypervisor，nova-compute 如何与它们配合呢？这就是我们之前讨论过的 Driver 架构。

nova-compute 为这些 Hypervisor 定义了统一的接口，Hypervisor 只需要实现这些接口，就可以 Driver 的形式即插即用 OpenStack 系统中。

如图 7-30 所示是 Nova Driver 的架构示意图。

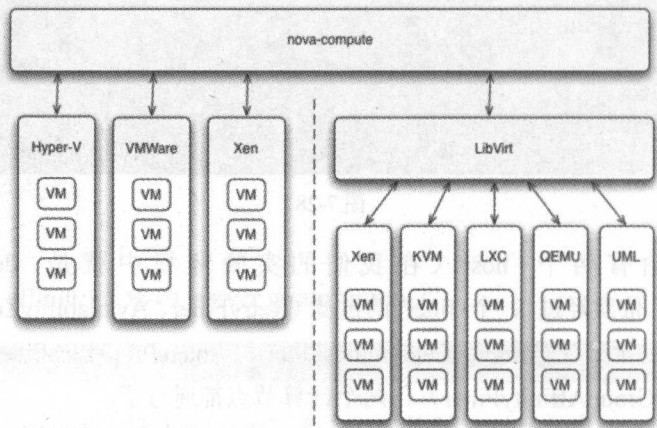


图 7-30

我们可以在 /opt/stack/nova/nova/virt/ 目录下查看到 OpenStack 源代码中已经自带了上面这几个 Hypervisor 的 Driver，如图 7-31 所示。

```
root@devstack-controller:~# ls -l /opt/stack/nova/nova/virt/ |grep "\.d"
drwxr-xr-x 4 stack stack 4096 Dec 10 20:45 disk
drwxr-xr-x 2 stack stack 4096 Dec 10 20:28 hyperv
drwxr-xr-x 2 stack stack 4096 Dec 10 20:45 image
drwxr-xr-x 2 stack stack 4096 Dec 10 20:45 ironic
drwxr-xr-x 4 stack stack 4096 Dec 10 20:45 libvirt
drwxr-xr-x 2 stack stack 4096 Dec 10 20:28 vmwareapi
drwxr-xr-x 4 stack stack 4096 Dec 10 20:28 xenapi
```

图 7-31

某个特定的计算节点上只会运行一种 Hypervisor，只需在该节点 nova-compute 的配置文件 /etc/nova/nova.conf 中配置所对应的 compute\_driver 就可以了。

在我们的环境中因为是 KVM，所以配置的是 Libvirt 的 driver，如图 7-32 所示。

```
compute_driver = libvirt.LibvirtDriver
```

图 7-32

nova-compute 的功能可以分为两类：

- 定时向 OpenStack 报告计算节点的状态。
- 实现 instance 生命周期的管理。

下面我们依次介绍。



## 2. 定期向 OpenStack 报告计算节点的状态

前面我们看到 nova-scheduler 的很多 Filter 是根据计算节点的资源使用情况进行过滤的。

比如 RamFilter 要检查计算节点当前可用的内存量；CoreFilter 检查可用的 vCPU 数量；DiskFilter 则会检查可用的磁盘空间。

那这里有个问题：OpenStack 是如何得知每个计算节点的这些信息呢？

答案就是：nova-compute 会定期向 OpenStack 报告。

从 nova-compute 的日志 /opt/stack/logs/n-cpu.log 可以发现：

每隔一段时间，nova-compute 就会报告当前计算节点的资源使用情况和 nova-compute 服务状态，如图 7-33 所示。

```
2016-01-04 17:07:00.220 INFO nova.compute.resource_tracker [req-ca809f79-d424-4f73-b8d7-38b5f331a23d None None] Final resource view: name=devstack-compute1 phys_ram=10000MB used_ram=512MB phys_disk=9GB used_disk=0GB total_vcpus=6 used_vcpus=0 pci_stats=None
2016-01-04 17:07:00.245 INFO nova.compute.resource_tracker [req-ca809f79-d424-4f73-b8d7-38b5f331a23d None None] Compute service record updated for devstack-compute1:devstack-compute1
```

图 7-33

如果我们再深入思考一个问题：nova-compute 是如何获得当前计算节点的资源使用信息的？给大家一分钟自己先思考一下？

好，揭晓答案。

要得到计算节点的资源使用详细情况，需要知道当前节点上所有 instance 的资源占用信息。这些信息谁最清楚？当然是 Hypervisor。

大家还记得之前我们讨论的 Nova Driver 架构吧，nova-compute 可以通过 Hypervisor 的 driver 拿到这些信息。

举例来说，在我们的实验环境下 Hypervisor 是 KVM，用的 Driver 是 LibvirtDriver。

LibvirtDriver 可以调用相关的 API 获得资源信息，这些 API 的作用相当于我们在 CLI 里执行 virsh nodeinfo、virsh domaininfo 等命令。

## 3. 实现 instance 生命周期的管理

OpenStack 对 instance 最主要的操作都是通过 nova-compute 实现的，包括 instance 的 launch、shutdown、reboot、suspend、resume、terminate、resize、migration、snapshot 等。

本小节重点学习 nova-compute 如何实现 instance launch（部署）操作，其他操作将会在后面的章节讨论。

当 nova-scheduler 选定了部署 instance 的计算节点后，会通过消息中间件 rabbitMQ 向选定的计算节点发出 launch instance 的命令。

该计算节点上运行的 nova-compute 收到消息后会执行 instance 创建操作。

日志 /opt/stack/logs/n-cpu.log 记录了整个操作过程。

nova-compute 创建 instance 的过程可以分为 4 步：

- 为 instance 准备资源。
- 创建 instance 的镜像文件。
- 创建 instance 的 XML 定义文件。
- 创建虚拟网络并启动虚拟机。

下面我们依次讨论每个步骤。

### (1) 为 instance 准备资源

nova-compute 首先会根据指定的 flavor 依次为 instance 分配内存、磁盘空间和 vCPU。可以在日志中看到这些细节，如图 7-34 所示。

```
2016-01-04 18:42:24.232 INFO nova.compute.manager [req-2d188ce6-1693-4858-8a02-621a14ea12fa admin admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] Starting instance...
2016-01-04 18:42:24.343 DEBUG oslo_concurrency.lockutils [req-2d188ce6-1693-4858-8a02-621a14ea12fa admin admin] Lock "compute_resources" acquired by "nova.compute.resource_tracker.instance_claim" :: waited 0.000s inner /usr/local/lib/python2.7/dist-packages/oslo_concurrency/lockutils.py:253
2016-01-04 18:42:24.344 DEBUG nova.compute.resource_tracker [req-2d188ce6-1693-4858-8a02-621a14ea12fa admin admin] Memory overhead for 64 MB instance; 0 MB instance_claim /opt/stack/nova/nova/compute/resource_tracker.py:170
2016-01-04 18:42:24.351 INFO nova.compute.claims [req-2d188ce6-1693-4858-8a02-621a14ea12fa admin admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] Attempting claim: memory, 64 MB, disk 0 GB
2016-01-04 18:42:24.351 INFO nova.compute.claims [req-2d188ce6-1693-4858-8a02-621a14ea12fa admin admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] Total memory: 10000 MB, used: 512.00 MB
2016-01-04 18:42:24.352 INFO nova.compute.claims [req-2d188ce6-1693-4858-8a02-621a14ea12fa admin admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] memory limit: 15000.00 MB, free: 14488.00 MB
2016-01-04 18:42:24.352 INFO nova.compute.claims [req-2d188ce6-1693-4858-8a02-621a14ea12fa admin admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] Total disk: 9 GB, used: 0.00 GB
2016-01-04 18:42:24.353 INFO nova.compute.claims [req-2d188ce6-1693-4858-8a02-621a14ea12fa admin admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] disk limit: 9.00 GB, free: 9.00 GB
2016-01-04 18:42:24.367 DEBUG nova.compute.resources.vcpu [req-2d188ce6-1693-4858-8a02-621a14ea12fa admin admin] Total CPUs: 6 vCPUs, used: 0.00 vCPUs test /opt/stack/nova/nova/compute/resources/vcpu.py:52
2016-01-04 18:42:24.367 DEBUG nova.compute.resources.vcpu [req-2d188ce6-1693-4858-8a02-621a14ea12fa admin admin] CPUs limit not specified, defaulting to unlimited test /opt/stack/nova/nova/compute/resources/vcpu.py:56
2016-01-04 18:42:24.368 INFO nova.compute.claims [req-2d188ce6-1693-4858-8a02-621a14ea12fa admin admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] Claim successful
```

图 7-34

网络资源也会提前分配，如图 7-35 所示。

```
2016-01-04 18:42:24.610.2880 DEBUG nova.compute.manager [-] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] Allocating IP information in the background. _allocate_network_async /opt/stack/nova/nova/compute/manager.py:1548
```

图 7-35

### (2) 创建 instance 的镜像文件

资源准备好之后，nova-compute 会为 instance 创建镜像文件。

OpenStack 启动一个 instance 时, 会选择一个 image, 这个 image 由 Glance 管理。nova-compute 会:

- 首先将该 image 下载到计算节点。
- 然后将其作为 backing file 创建 instance 的镜像文件。
- 从 Glance 下载 image。

nova-compute 首先会检查 image 是否已经下载(比如之前已经创建过基于相同 image 的 instance)。如果没有, 就从 Glance 下载 image 到本地。

由此可知, 如果计算节点上要运行多个相同 image 的 instance, 只会在启动第一个 instance 的时候从 Glance 下载 image, 后面的 instance 启动速度就大大加快了。

日志如图 7-36 所示。

```
2016-01-04 18:42:25.034 DEBUG nova.virt.images [req-2d188ce6-1693-48
58-8a02-621a14ea12fa admin admin] 917d60ef-f663-4e2d-b85b-e4511bb56b
c2 was qcow2, converting to raw, fetch_to_raw /opt/stack/nova/nova/vi
rt/images.py:124
2016-01-04 18:42:25.035 DEBUG oslo_concurrency.processutils [req-2d1
88ce6-1693-4858-8a02-621a14ea12fa admin admin] Running cmd (subproce
ss): qemu-img convert -O raw /opt/stack/data/nova/instances/_base/60
bba5916c6c90ed2ef7d3263de8f653111dd35f.part /opt/stack/data/nova/ins
tances/_base/60bba5916c6c90ed2ef7d3263de8f653111dd35f.converted exec
ute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processu
tils.py:250
```

图 7-36

从日志中可以看到:

- image (ID 为 917d60ef-f663-4e2d-b85b-e4511bb56bc2) 是 qcow2 格式, nova-compute 将其下载, 然后通过 qemu-img 转换成 raw 格式。转换的原因是下一步需要将其作为 instance 的镜像文件的 backing file, 而 backing file 不能是 qcow2 格式。
- image 的存放目录是 /opt/stack/data/nova/instances/\_base, 这是由 /etc/nova/nova.conf 的下面两个配置选项决定的。

```
instance_spath = /opt/stack/data/nova/instances
base_dir_name = _base
```

- 下载的 image 文件被命名为 60bba5916c6c90ed2ef7d3263de8f653111dd35f, 这是 image id 的 SHA1 哈希值。
- 为 instance 创建镜像文件。

有了 image 之后, instance 的镜像文件直接通过 qemu-img 命令创建, backing file 就是下载的 image, 如图 7-37 所示。

```
2016-01-04 18:42:25.294 DEBUG oslo_concurrency.processutils [req-2d188ce6-1693-4858-8a02-621a14ea12fa admin admin] Running cmd (subprocess): [qemu-img create -f qcow2 -o backing_file=/opt/stack/data/nova/instances/_base/60bba5916c6c90ed2ef7d3263de8f653111dd35f /opt/stack/data/nova/instances/f1e22596-6844-4d7a-84a3-e41e6d7618ef/disk execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250]
```

图 7-37

这里 instance 的镜像文件位于 `/opt/stack/data/nova/instances/f1e22596-6844-4d7a-84a3-e41e6d7618ef/disk`，格式为 `qcow2`，其中 `f1e22596-6844-4d7a-84a3-e41e6d7618ef` 就是 instance 的 ID。

可以通过 `qemu-info` 查看 disk 文件的属性，如图 7-38 所示。

```
root@devstack-controller:~# qemu-img info /opt/stack/data/nova/instances/f1e22596-6844-4d7a-84a3-e41e6d7618ef/disk
image: /opt/stack/data/nova/instances/f1e22596-6844-4d7a-84a3-e41e6d7618ef/disk
file format: qcow2
virtual size: 39M (41126400 bytes)
disk size: 1.6M
cluster_size: 65536
backing file: /opt/stack/data/nova/instances/_base/60bba5916c6c90ed2ef7d3263de8f653111dd35f
Format specific information:
  compat: 1.1
  lazy refcounts: false
```

图 7-38

这里有两个容易搞混淆的术语，在此特别说明一下：

- image，指的是 Glance 上保存的镜像，作为 instance 运行的模板。计算节点将下载的 image 存放在 `/opt/stack/data/nova/instances/_base` 目录下。
- 镜像文件，指的是 instance 启动盘所对应的文件。
- 二者的关系是：image 是镜像文件的 backing file。image 不会变，而镜像文件会发生变化。比如安装新的软件后，镜像文件会变大。

因为英文中两者都叫“image”，为避免混淆，我们用“image”和“镜像文件”作区分。

### (3) 创建 instance 的 XML 定义文件

创建 instance 的 XML 定义文件，如图 7-39 所示。

```
2016-01-04 18:42:25.595 DEBUG nova.virt.libvirt.config [req-2d188ce6-1693-4858-8a02-621a14ea12fa admin admin] Generated XML (<domain type="qemu">\n  <uuid>f1e22596-6844-4d7a-84a3-e41e6d7618ef</uuid>\n  <name>instance-00000002</name>\n  <memory>65536</memory>\n  <vcpu>1</vcpu>\n  <metadata>\n    <nova:instance xmlns:nova="http://openstack.org/xmlns/libvirt/nova/1.0">\n      <nova:package version="12.0.1">\n        <nova:name>c2</nova:name>\n      </nova:package>\n    </nova:instance>\n  </metadata>\n</domain>\n  <nova:creationTime>2016-01-04 18:42:25.595</nova:creationTime>\n</domain>
```

图 7-39

创建的 XML 文件会保存到该 instance 目录 `/opt/stack/data/nova/instances/f1e22596-6844-4d7a-84a3-e41e6d7618ef`，命名为 `libvirt.xml`。



## (4) 创建虚拟网络并启动 instance

接下来便是为 instance 创建虚拟网络设备, 如图 7-40 所示。

```
2016-01-04 18:42:25.599 DEBUG nova.virt.libvirt.vif [req-2d188ce6-1093-4858-8a02-621a14ea12fa admin:admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] Ensuring bridge brq5333bf1-67 plug_bridge /opt/stack/nova/nova/virt/libvirt/vif.py:482
```

图 7-40

本环境用的是 linux-bridge 实现的虚拟网络, 在 Neutron 章节我们会详细讨论 OpenStack 虚拟网络的不同实现方式。

一切就绪, 接下来可以启动 instance 了, 如图 7-41 和图 7-42 所示。

```
2016-01-04 18:42:26.339.2880 INFO nova.compute.manager [-] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] VM Started (Lifecycle Event)
2016-01-04 18:42:26.391 DEBUG nova.compute.manager [req-c787cd00-d5cd-4d85-9a26-b247dcb6f90c None:None] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] checking state _get_power_state /opt/stack/nova/nova/compute/manager.py:1331
2016-01-04 18:42:26.396 DEBUG nova.virt.driver [req-c787cd00-d5cd-4d85-9a26-b247dcb6f90c None:None] Emitting event <LifecycleEvent: 1451904146.34, f1e22596-6844-4d7a-84a3-e41e6d7618ef => Paused> emit_event /opt/stack/nova/nova/virt/driver.py:1303
2016-01-04 18:42:26.396 INFO nova.compute.manager [req-c787cd00-d5cd-4d85-9a26-b247dcb6f90c None:None] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] VM Paused (Lifecycle Event)
2016-01-04 18:42:26.445 DEBUG nova.compute.manager [req-c787cd00-d5cd-4d85-9a26-b247dcb6f90c None:None] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] checking state _get_power_state /opt/stack/nova/nova/compute/manager.py:1331
2016-01-04 18:42:26.450 DEBUG nova.compute.manager [req-c787cd00-d5cd-4d85-9a26-b247dcb6f90c None:None] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] synchronizing instance power state after lifecycle event "Paused"; current vm state: building; current task state: spawning; current DB power state: 0; vm power state: 3 handle_lifecycle_event /opt/stack/nova/nova/compute/manager.py:1260
2016-01-04 18:42:26.507 INFO nova.compute.manager [req-c787cd00-d5cd-4d85-9a26-b247dcb6f90c None:None] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] during sync power state the instance has a pending task (spawning). Skip.
```

图 7-41

```
2016-01-04 18:42:31.241 INFO nova.compute.manager [req-c787cd00-d5cd-4d85-9a26-b247dcb6f90c None:None] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] VM Resumed (Lifecycle Event)
2016-01-04 18:42:31.243 DEBUG nova.virt.libvirt.driver [req-2d188ce6-1693-4858-8a02-621a14ea12fa admin:admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] Instance is running spawn /opt/stack/nova/nova/virt/libvirt/driver.py:2445
2016-01-04 18:42:31.248 2880 INFO nova.virt.libvirt.driver [-] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] Instance spawned successfully.
```

图 7-42

至此, instance 已经成功启动。

OpenStack 图形界面和 KVM CLI 都可以查看到 instance 的运行状态, 如图 7-43 所示。

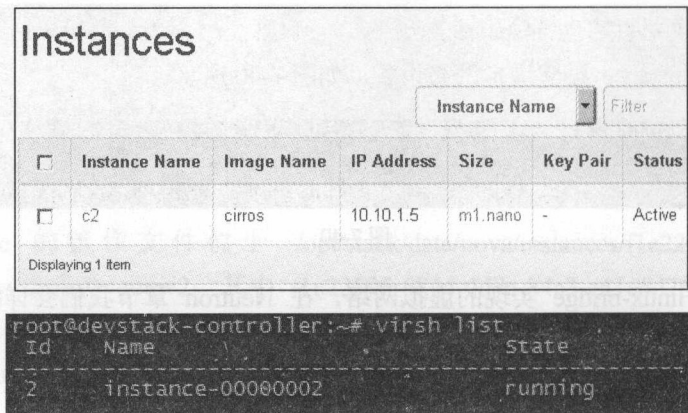


图 7-43

在计算节点上, instance 并不是以 OpenStack 上的名字命名, 而是用 instance-xxxxx 的格式。

## 7.2.4 nova-conductor

nova-compute 需要获取和更新数据库中 instance 的信息。

但 nova-compute 并不会直接访问数据库, 而是通过 nova-conductor 实现数据的访问, 如图 7-44 所示。

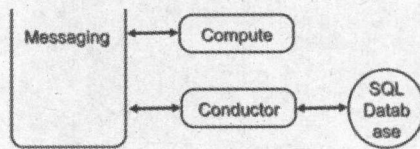


图 7-44

这样做有两个显著好处:

- 更高的系统安全性。
- 更好的系统伸缩性。

### 1. 更高的安全性

在 OpenStack 的早期版本中, nova-compute 可以直接访问数据库, 但这样存在非常大的安全隐患。

因为 nova-compute 这个服务是部署在计算节点上的, 为了能够访问控制节点上的数据库, 就必须在计算节点的 /etc/nova/nova.conf 中配置访问数据库的连接信息, 比如:

```
[database]
connection = mysql+pymysql://root:secret@controller/nova?charset=utf8
```

试想任意一个计算节点被黑客入侵, 都会导致部署在控制节点上的数据库面临极大风险。

为了解决这个问题, 从 G 版本开始, Nova 引入了一个新服务 nova-conductor, 将

nova-compute 访问数据库的全部操作都放到 nova-conductor 中，而且 nova-conductor 是部署在控制节点上的。

这样就避免了 nova-compute 直接访问数据库，增加了系统的安全性。

## 2. 更好的伸缩性

nova-conductor 将 nova-compute 与数据库解耦之后还带来另一个好处：提高了 nova 的伸缩性。

nova-compute 与 conductor 是通过消息中间件交互的。

这种松散的架构允许配置多个 nova-conductor 实例。

在一个大规模的 OpenStack 部署环境里，管理员可以通过增加 nova-conductor 的数量来应对日益增长的计算节点对数据库的访问。

# 7.3 通过场景学习 Nova

instance 从创建到删除的整个生命周期都是由 Nova 管理的。

后面各小节我们以 instance 生命周期中的不同操作场景为例，详细分析 Nova 不同组件如何协调工作，并通过日志分析加深大家对 Nova 的理解。

## 7.3.1 看懂 OpenStack 日志

在研究 Nova 各个操作之前，我们先来学习一个重要的内容：OpenStack 日志。

OpenStack 的日志记录了非常详细的细节信息，是我们学习和 troubleshooting 的利器。

### 1. 日志的位置

我们实验环境使用的是 devstack，日志都统一放在 /opt/stack/logs 目录下，每个服务有自己的日志文件，从命名上很容易区分，如 7-45 图所示。

```
root@devstack-controller:~# ls /opt/stack/logs
c-api.log
c-api.log.2015-12-10-195926
c-sch.log
c-sch.log.2015-12-10-195926
c-vol.log
c-vol.log.2015-12-10-195926
g-api.log
g-api.log.2015-12-10-195926
g-reg.log
g-reg.log.2015-12-10-195926
horizon.log
horizon.log.2015-12-10-195926
key-access.log
key-access.log.2015-12-10-195926
key.log
key.log.2015-12-10-195926
n-api.log
n-api.log.2015-12-10-195926
n-cauth.log
n-cauth.log.2015-12-10-195926
n-cond.log
n-cond.log.2015-12-10-195926
n-cpu.log
n-cpu.log.2015-12-10-195926
n-ert.log
n-ert.log.2015-12-10-195926
n-novnc.log
n-novnc.log.2015-12-10-195926
n-sch.log
n-sch.log.2015-12-10-195926
q-agt.log
q-agt.log.2015-12-10-195926
q-dhcp.log
q-dhcp.log.2015-12-10-195926
q-l3.log
q-l3.log.2015-12-10-195926
q-meta.log
q-meta.log.2015-12-10-195926
q-svc.log
q-svc.log.2015-12-10-195926
```

图 7-45

比如 nova-\* 各个子服务的日志都以“n-”开头：

- n-api.log 是 nova-api 的日志。
- n-cpu.log 是 nova-compute 的日志。

Glance 的日志文件都是“g-”开头：

- g-api.log 是 glance-api 的日志。
- g-reg.log 是 glance-registry 的日志。

Cinder、Neutron 的日志分别以“c-”和“q-”开头。

对于非 devstack 安装的 OpenStack，日志一般放在 /var/log/xxx/ 目录下。

比如 Nova 放在 /var/log/nova/ 下，Glance 放在 /var/log/glance 下等。

各个子服务的日志文件也是单独保存，命名也很规范，容易区分。

比如 nova-api 的日志一般就命名为 /var/log/nova/api.log，其他日志类似。

## 2. 日志的格式

OpenStack 的日志格式都是统一的，如下：

```
<时间戳><日志等级> <Request ID><日志内容><源代码位置>
```

简单说明一下。

- 时间戳，日志记录的时间，包括 年 月 日 时 分 秒 毫秒。
- 日志等级，有 INFO WARNING ERROR DEBUG 等。
- 代码模块，当前运行的模块。
- Request ID，日志会记录连续不同的操作，为了便于区分和增加可读性，每个操作都被分配唯一的 Request ID，便于查找。
- 日志内容，这是日志的主体，记录当前正在执行的操作和结果等重要信息。
- 源代码位置，日志代码的位置，包括方法名称，源代码文件的目录位置和行号。这一项不是所有日志都有。

下面举例说明。

```
2015-12-10 20:46:49.566 DEBUG nova.virt.libvirt.config
[req-5c973fff-e9ba-4317-bfd9-76678cc96584 None None] Generated XML ('<cpu>
<arch>x86_64</arch><model>Westmere</model><vendor>Intel</vendor><topolog
y sockets="2" cores="3" threads="1"/><feature name="avx"/><feature
name="ds"/><feature name="ht"/><feature name="hypervisor"/><feature
name="osxsave"/><feature name="pclmuldq"/><feature name="rdtsclp"/><feature
name="ss"/><feature name="vme"/><feature name="xsave"/></cpu>\n',) to_xml
/opt/stack/nova/nova/virt/libvirt/config.py:82
```



从这条日志我们可以得知：

- 代码模块是 `nova.virt.libvirt.config`，由此可知应该是 Hypervisor Libvirt 相关的操作。
- 日志内容是生成 XML。
- 如果要跟踪源代码，可以到 `/opt/stack/nova/nova/virt/libvirt/config.py` 的 82 行，方法是 `to_xml`，如图 7-46 所示。

```
79 def to_xml(self, pretty_print=True):
80     root = self.format_dom()
81     xml_str = etree.tostring(root, pretty_print=pretty_print)
82     LOG.debug('XML string: %s', (xml_str,))
83     return xml_str
```

图 7-46

又例如下面这条日志：

```
2015-12-10 20:46:49.671 ERROR nova.compute.manager
[req-5c973ffff-e9ba-4317-bfd9-76678cc96584 None None] No compute node record
for host devstack-controller
```

从这条日志我们可以得知：

- 这是一个 ERROR 日志。
- 具体内容是“No compute node record for host devstack-controller”。
- 该日志没有指明源代码位置。

### 3. 关于日志的几点说明

学习 OpenStack 需要看日志吗？这个问题的答案取决于你是谁。

如果你只是 OpenStack 的最终用户，那么日志对你不重要。你只需要在 GUI 上操作，如果出问题直接找管理员就可以了。

但如果你是 OpenStack 的运维和管理人员，日志对你就非常重要了。因为 OpenStack 操作如果出错，GUI 上给出的错误信息是非常笼统和简要的，日志则提供了大量的线索，特别是当 debug 选项打开之后。

如果你正处于 OpenStack 的学习阶段，正如我们现在的状态，那么也强烈建议你多看日志。日志能够帮助你更加深入理解 OpenStack 的运行机制。

日志能够帮助我们深入学习 OpenStack 和排查问题。但要想高效地使用日志还得有个前提：必须先掌握 OpenStack 的运行机制，然后针对性地查看日志。就拿 Instance Launch 操作来说，如果之前不了解 nova-\* 各子服务在操作中的协作关系，如果没有理解流程图，面对如此多和分散的日志文件，我们也很难下手。

对于 OpenStack 的运维和管理员来说，在大部分情况下，我们都不需要看源代码。因为 OpenStack 的日志记录得很详细了，足以帮助我们分析和定位问题。但还是有一些细节日志没有记录，必要时可以通过查看源代码理解得更清楚。即便如此，日志也会为我们提供源代码查看的

线索，不需要我们大海捞针。这一点我们会在后面的操作分析中看到。

## 7.3.2 Launch

Launch instance 应该算是 Nova 最重要的操作。

仔细研究 launch 操作能够帮助我们充分理解 Nova 各个子服务的协调配合和运行机制。

前面我们已经以 launch 操作为例详细讨论了各个 nova-\* 子服务。这里不再赘述，只是再回顾一下流程，如图 7-47 所示。

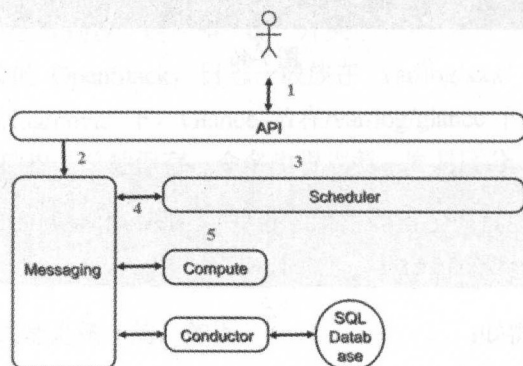


图 7-47

- 客户（可以是 OpenStack 最终用户，也可以是其他程序）向 API (nova-api) 发送请求：“帮我创建一个 Instance”。
- API 对请求做一些必要处理后，向 Messaging (RabbitMQ) 发送了一条消息：“让 Scheduler 创建一个 Instance”。
- Scheduler (nova-scheduler) 从 Messaging 获取到 API 发给它的消息，然后执行调度算法，从若干计算节点中选出节点 A。
- Scheduler 向 Messaging 发送了一条消息：“在计算节点 A 上创建这个 Instance”。
- 计算节点 A 的 Compute (nova-compute) 从 Messaging 中获取到 Scheduler 发给它的消息，然后通过本节点的 Hypervisor Driver 创建 Instance。
- 在 Instance 创建的过程中，Compute 如果需要查询或更新数据库信息，会通过 Messaging 向 Conductor (nova-conductor) 发送消息，Conductor 负责数据库访问。

## 7.3.3 Shut Off

如图 7-48 所示是 shut off instance 的流程图。

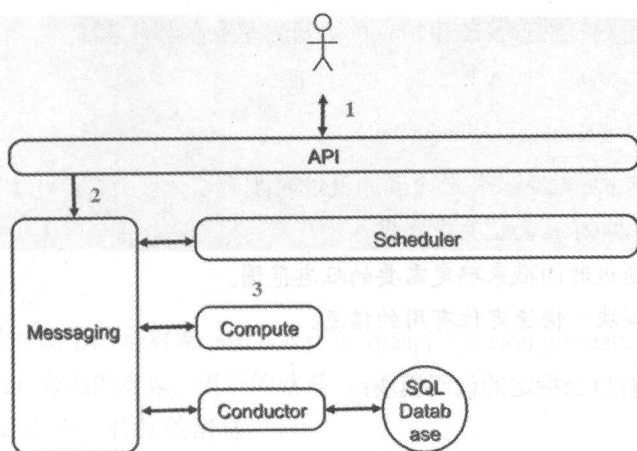


图 7-48

- 向 nova-api 发送请求。
- nova-api 发送消息。
- nova-compute 执行操作。

下面我们详细讨论每一个步骤。

### 1. 向 nova-api 发送请求

客户（可以是 OpenStack 最终用户，也可以是其他程序）向 API（nova-api）发送请求：“帮我关闭这个 Instance”，如图 7-49 所示。

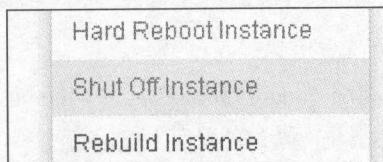


图 7-49

查看日志 /opt/stack/logs/n-api.log，如图 7-50 所示。

```

2016-01-06 17:03:59.426 DEBUG nova.api.openstack.compute.servers
[req-1758b389-a2d0-44cc-a95a-6f75e4dc07fd admin admin] [instance:
f1e22596-6844-4d7a-84a3-e41e6d7618ef] stop_instance stop_server
/opt/stack/nova/nova/api/openstack/compute/servers.py:1158
2016-01-06 17:03:59.427 DEBUG nova.compute.api [req-1758b389-a2d0-44cc-a95a-6f75e4dc07fd admin admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] Going to try to stop instance force_stop /opt/stack/nova/nova/compute/api.py:1977
2016-01-06 17:03:59.505 INFO nova.osapi_compute.wsgi.server [req-1758b389-a2d0-44cc-a95a-6f75e4dc07fd admin admin] 192.168.104.10:8080 POST /v2.1/aa81b851d2a54484b6f3984ab2c5d4e4/servers/f1e22596-6844-4d7a-84a3-e41e6d7618ef/action HTTP/1.1" status: 202 len: 272 time: 0.1760890
  
```

图 7-50

对于如何在日志文件中快速查找有用的信息这里多聊几句。

对于初学者，这不是一件容易的事情，因为日志里条目和内容很多，特别是 debug 选项打开之后，容易让人眼花缭乱，无从下手。

这里给大家几个小窍门：

- 先确定大的范围，比如在操作之前用 `tail -f` 打印日志文件，这样需要查看的日志肯定在操作之后的打印输出的这些内容里。
- 另外也可以通过时间戳来确定需要的日志范围。
- 利用“代码模块”快速定位有用的信息。

nova-\* 子服务都有自己特定的代码模块：

- nova-api

```
nova.api.openstack.compute.servers
nova.compute.api
nova.api.openstack.wsgi
```

- nova-compute

```
nova.compute.manager
nova.virt.libvirt.*
```

- nova-scheduler

```
nova.scheduler.*
```

利用 Request ID 查找相关的日志信息。

在上面的日志中，我们可以利用“req-1758b389-a2d0-44cc-a95a-6f75e4dc07fd”这个 Request ID 快速定位 n-api.log 中相与 shut off 操作的其他日志条目。

需要补充说明的是，Request ID 是跨日志文件的，这一个特性能帮助我们在其他子服务的日志文件中找到相关信息，我们后面马上将会看到这个技巧的应用。

## 2. nova-api 发送消息

nova-api 向 Messaging (RabbitMQ) 发送了一条消息：“关闭这个 Instance”

nova-api 没有将发送消息的操作记录到日志中，不过我们可以通过查看源代码来验证。

一提到源代码，大家可能以为要大海捞针了。其实很简单，上面日志已经清楚地告诉我们需要查看的源代码在 `/opt/stack/nova/nova/compute/api.py` 的 1977 行，方法是 `force_stop`，如图 7-51 所示。



```

1976     def force_stop(self, context, instance, do_cast=True, clean_shutdown=True):
1977         LOG.debug('force_stop: instance=%s', instance.name, instance=instance)
1978
1979         instance.task_state = task_states.POWERING_OFF
1980         instance.progress =
1981         instance.save(expected_task_state=[None])
1982
1983         self._record_action_start(context, instance, instance_actions.STOP)
1984
1985         self._compute_rpcapi.stop_instance(context, instance, do_cast=do_cast,
1986                                           clean_shutdown=clean_shutdown)
1987

```

图 7-51

forcestop 方法最后调用的是对象 self.compute\_rpcapi 的 stop\_instance 方法。在 OpenStack 源码中，以 xxx\_rpcapi 命名的对象，表示的就是 xxx 的消息队列。xxx\_rpcapi.yyy() 方法则表示向 xxx 的消息队列发送 yyy 操作的消息。

所以 self.compute\_rpcapi.stop\_instance() 的作用就是向 RabbitMQ 上 nova-compute 的消息队列里发送一条 stop\_instance 的消息。

这里补充说明一下：关闭 instance 的前提是 instance 当前已经在某个计算节点上运行，所以这里不需要 nova-scheduler 再帮我们挑选合适的节点，这个跟 launch 操作不同。

### 3. nova-compute 执行操作

查看计算节点上的日志 /opt/stack/logs/n-cpu.log，如图 7-52 所示。

```

2016-01-06 17:03:59.535 DEBUG nova.compute.manager [req-1758b389-
a2d0-44cc-a95a-6f75e4dc07fd,admin,admin] [instance: f1e22596-6844-
4d7a-84a3-e41e6d7618ef] Stopping instance; current vm_state: act
ive, current task_state: powering-off, current DB power_state: 1,
current VM power_state: 1 do_stop_instance /opt/stack/nova/nova/
compute/manager.py:2445
2016-01-06 17:03:59.536 DEBUG nova.objects.instance [req-1758b389-
a2d0-44cc-a95a-6f75e4dc07fd,admin,admin] Lazy-loading 'flavor' o
n instance uuid f1e22596-6844-4d7a-84a3-e41e6d7618ef obj_load_att
r /opt/stack/nova/nova/objects/instance.py:860
2016-01-06 17:03:59.611 DEBUG nova.virt.libvirt.driver [req-1758b389-
a2d0-44cc-a95a-6f75e4dc07fd,admin,admin] [instance: f1e22596-
6844-4d7a-84a3-e41e6d7618ef] Shutting down instance from state 1
_clean_shutdown /opt/stack/nova/nova/virt/libvirt/driver.py:2227
2016-01-06 17:04:02.631 INFO nova.virt.libvirt.driver [req-1758b389-
a2d0-44cc-a95a-6f75e4dc07fd,admin,admin] [instance: f1e22596-6844-
4d7a-84a3-e41e6d7618ef] Instance shutdown successfully after
3 seconds.

```

图 7-52

这里我们利用了 Request ID “req-1758b389-a2d0-44cc-a95a-6f75e4dc07fd”，在 n-cpu.log 中快速定位到 nova-compute 关闭 instance 的日志条目。

### 4. 小结

分析某个操作时，我们首先要理清该操作的内部流程，然后再到相应的节点上去查看日志。例如 shut off 的流程为：

- 向 nova-api 发送请求。
- nova-api 发送消息。

- nova-compute 执行操作。

第 1、2 两个步骤是在控制节点上执行的，查看 nova-api 的日志。第 3 步是在计算节点上执行的，查看 nova-compute 的日志。

### 7.3.4 Start

如图 7-53 所示是 start instance 的流程图。

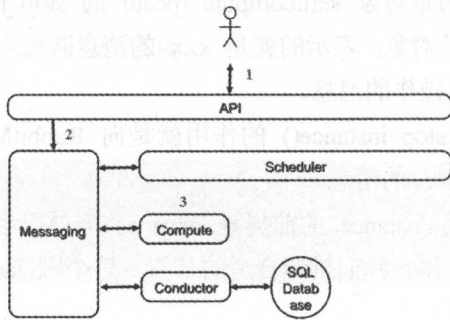


图 7-53

- 向 nova-api 发送请求。
- nova-api 发送消息。
- nova-compute 执行操作。

下面我们详细讨论每一个步骤。

#### 1. 向 nova-api 发送请求

客户（可以是 OpenStack 最终用户，也可以是其他程序）向 API（nova-api）发送请求：“帮我启动这个 Instance”，如图 7-54 所示。

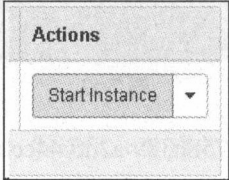


图 7-54

查看日志 /opt/stack/logs/n-api.log，如图 7-55 所示。

```
2016-01-05 18:38:16.336 DEBUG nova.api.openstack.compute.servers [req-246e2ce2-4af2-4c56-9020-ca2269a1b984 admin admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] start_instance _start_server /opt/stack/nova/nova/api/openstack/compute/servers.py:1139
2016-01-05 18:38:16.336 DEBUG nova.compute.api [req-246e2ce2-4af2-4c56-9020-ca2269a1b984 admin admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef]. Going to try to start instance start /opt/stack/nova/nova/compute/api.py:2002
```

图 7-55

## 2. nova-api 发送消息

nova-api 向 Messaging (RabbitMQ) 发送了一条消息：“启动这个 Instance”

查看源代码 /opt/stack/nova/nova/compute/api.py 的 2002 行，方法是 start，如图 7-56 所示。

```

2000     def start(self, context, instance):
2001
2002         LOG.debug('start instance %s', instance=instance)
2003
2004         instance.task_state = task_states.POWERING_ON
2005         instance.save(expected_task_state=[None])
2006
2007         self._record_action_start(context, instance, instance_actions.START)
2008         LOG.info('Starting instance %s', instance=instance)
2009
2010         self._compute_rpcapi.start_instance(context, instance)
2011
2012

```

图 7-56

self.compute\_rpcapi.start\_instance() 的作用就是向 RabbitMQ 上 nova-compute 的消息队列里发送一条 start\_instance 的消息。

## 3. nova-compute 执行操作

查看日志 /opt/stack/logs/n-cpu.log

- 开始启动，如图 7-57 所示。

```

2016-01-05 18:38:16.925 DEBUG nova.virt.libvirt.driver [req-246e2ce2-4af2-4c56-9020-ca2269a1b984 admin admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] [start] _get_guest_xml network_info=[VIF({'profile': {}, 'ovs_interfaceid': None, 'preserve_on_delete': False, 'network': Network({'bridge': u'brqas333bf1-67', 'subnets': [Subnet({'ip

```

图 7-57

- 准备虚拟网卡，如图 7-58 所示。

```

2016-01-05 18:38:17.068 DEBUG nova.virt.libvirt.vif [req-246e2ce2-4af2-4c56-9020-ca2269a1b984 admin admin] vif_type=bridge instance=instance(access_ip_v4=None;access_ip_v6=None;architecture=None;auto_disk

```

图 7-58

- 准备 instance 的 XML 文件，如图 7-59 所示。

```

2016-01-05 18:38:17.071 DEBUG nova.virt.libvirt.config [req-246e2ce2-4af2-4c56-9020-ca2269a1b984 admin admin] Generated XML (<domain type="qemu">\n  <uuid>f1e22596-6844-4d7a-84a3-e41e6d7618ef</uuid>\n  <name>instance-00000002</name>\n  <memory>65536</memory>\n  <vcpu>1<

```

图 7-59

- 准备 instance 镜像文件，如图 7-60 所示。

```

2016-01-05 18:38:17.073 DEBUG oslo_concurrency.processutils [req-246e2ce2-4af2-4c56-9020-ca2269a1b984 admin admin] Running cmd (subprocess): env LC_ALL=C LANG=C /qemu-img info /opt/stack/data/nova/instance-f1e22596-6844-4d7a-84a3-e41e6d7618ef/disk execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250

```

图 7-60

- 成功启动，如图 7-61 所示。

```

2016-01-05 18:38:17.872 INFO nova.compute.manager [req-c787cd00-d5cd
-4d85-9a26-b247dcb6f90c None None] [instance: fle22596-6844-4d7a-84a
3-e41e6d7618ef] VM Started (Lifecycle Event)
2016-01-05 18:38:17.881 2880 INFO nova.virt.libvirt.driver [-] [inst
ance: fle22596-6844-4d7a-84a3-e41e6d7618ef] Instance rebooted succe
sfully.

```

图 7-61

### 7.3.5 Soft/Hard Reboot

有两种启动方式 soft reboot 与 hard reboot, 如图 7-62 所示。

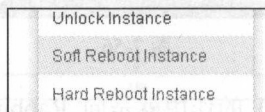


图 7-62

soft reboot 与 hard reboot 的区别在于:

(1) soft reboot 只是重启操作系统, 整个过程中, instance 依然处于运行状态。相当于在 Linux 中执行 reboot 命令。

(2) hard reboot 是重启 instance, 相当于关机之后再开机。soft/hard reboot 的日志分析留给大家作为练习。

(3) soft/hard reboot 在 nova-api 的日志里找不到, 这是因为 /opt/stack/nova/nova/compute/api.py 的 reboot 方法中没有输出 log。可以通过关键字 “nova.api.openstack.wsgi” 或者 “reboot” 搜索。

(4) 在 nova-compute 的日志中可以看到 “soft reboot” 和 “hard reboot” 二者有明显的区别。

### 7.3.6 Lock/Unlock

为了避免误操作, 比如意外重启或删除 instance, 可以将 instance 加锁。对被加锁 (Lock) 的 instance 执行重启等改变状态的操作会提示操作不允许。

执行解锁 (Unlock) 操作后恢复正常。

Lock/Unlock 操作都是在 nova-api 中进行的。操作成功后 nova-api 会更新 instance 加锁的状态。执行其他操作时, nova-api 根据加锁状态来判断是否允许。

Lock/Unlock 不需要 nova-compute 的参与。

Lock/Unlock 的日志比较简单, 留给大家练习。

提示:

(1) admin 角色的用户不受 Lock 的影响, 及无论加锁与否都可以正常执行操作。

(2) 根据默认 policy 的配置, 任何用户都可以 Unlock。也就是说如果发现 instance 被加锁了, 可以通过 Unlock 解锁, 然后再执行操作。



### 7.3.7 Terminate

Terminate 操作就是删除 instance，如图 7-63 所示是 terminate instance 的流程图。

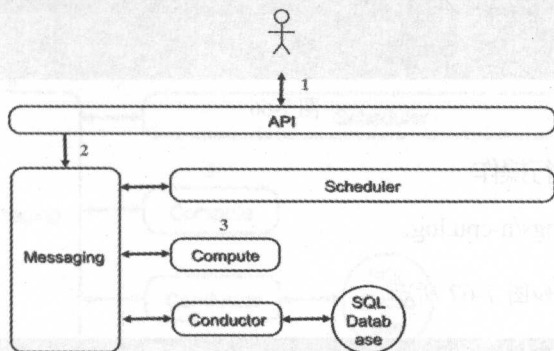


图 7-63

- 向 nova-api 发送请求。
- nova-api 发送消息。
- nova-compute 执行操作。

下面我们详细讨论每一个步骤。

#### 1. 向 nova-api 发送请求

客户（可以是 OpenStack 最终用户，也可以是其他程序）向 API（nova-api）发送请求：“帮我删除这个 Instance”，如图 7-64 所示。

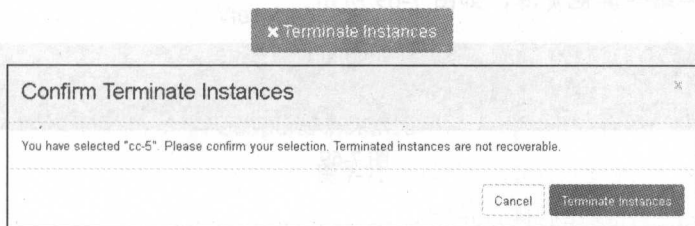


图 7-64

查看日志 `/opt/stack/logs/n-api.log`，如图 7-65 所示。

```

2016-01-17 17:24:24.237 DEBUG nova.compute.api [req-55ec8c2a-1487-4835-b164-7ec9e17c5942 admin:admin] [instance: 4e828b6e-2f84-44c1-bdd0-834c76ade6a1]
Going to try to terminate instance delete /opt/stack/nova/nova/compute/api
py:1934
  
```

图 7-65

#### 2. nova-api 发送消息

nova-api 向 Messaging（RabbitMQ）发送了一条消息：“删除这个 Instance”。源代码在 `/opt/stack/nova/nova/compute/api.py`，方法是 `do_force_delete`，如图 7-66 所示。

```

1885     def _do_force_delete(self, context, instance, bdms, reservations=None,
1886                          local=False):
1887         if local:
1888             instance.vm_state = vm_states.DELETED
1889             instance.task_state = None
1890             instance.terminated_at = timeutils.utcnow()
1891             instance.save()
1892         else:
1893             self.compute_rpcapi.terminate_instance(context, instance, bdms,
1894                                                     reservations=reservations,
1895                                                     delete_type="force")

```

图 7-66

### 3. nova-compute 执行操作

查看日志 /opt/stack/logs/n-cpu.log。

- 关闭 instance，如图 7-67 所示。

```

2016-01-17 17:24:24.394 INFO nova.compute.manager [req-55ec8c2a-1487-4835-b
164-7ec9e17c5942 admin:admin] [instance: 4e828b6e-2f84-44c1-bdd0-834c76ade6
a1] Terminating instance
2016-01-17 17:24:24.915.2880 INFO nova.virt.libvirt.driver [-] [instance: 4
e828b6e-2f84-44c1-bdd0-834c76ade6a1] Instance destroyed successfully.

```

图 7-67

- 删除 instance 的镜像文件，如图 7-68 所示。

```

2016-01-17 17:24:24.938 INFO nova.virt.libvirt.driver [req-55ec8c2a-1487-48
35-b164-7ec9e17c5942 admin:admin] [instance: 4e828b6e-2f84-44c1-bdd0-834c76
ade6a1] Deleting instance files /opt/stack/data/nova/instances/4e828b6e-2f8
4-44c1-bdd0-834c76ade6a1 del
2016-01-17 17:24:24.939 INFO nova.virt.libvirt.driver [req-55ec8c2a-1487-48
35-b164-7ec9e17c5942 admin:admin] [instance: 4e828b6e-2f84-44c1-bdd0-834c76
ade6a1] Deletion of /opt/stack/data/nova/instances/4e828b6e-2f84-44c1-bdd0-
834c76ade6a1 del complete

```

图 7-68

- 释放虚拟网络等其他资源，如图 7-69 所示。

```

2016-01-17 17:24:25.056 DEBUG nova.compute.manager [req-55ec8c2a-1487-4835-
b164-7ec9e17c5942 admin:admin] [instance: 4e828b6e-2f84-44c1-bdd0-834c76ade
6a1] Deallocating network for instance _deallocate_network /opt/stack/nova/
nova/compute/manager.py:1808

```

图 7-69

## 7.3.8 Pause/Resume

有时需要短时间暂停 instance，可以通过 Pause 操作将 instance 的状态保存到宿主机的内存中。当需要恢复的时候，执行 Resume 操作，从内存中读回 instance 的状态，然后继续运行 instance。

如图 7-70 所示是 pause instance 的流程图。

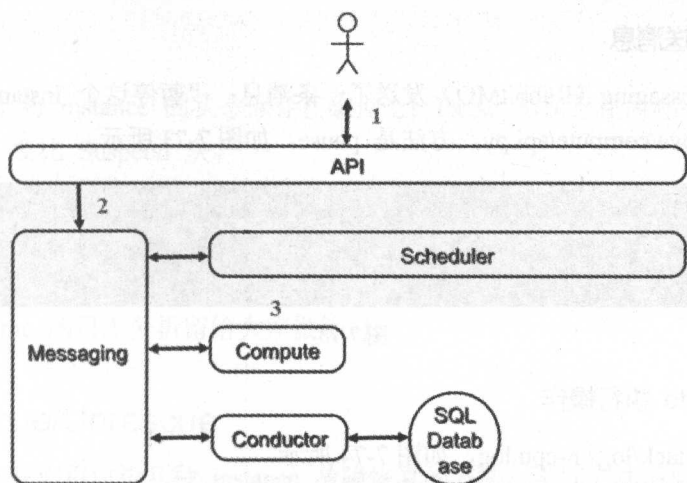


图 7-70

- 向 nova-api 发送请求。
- nova-api 发送消息。
- nova-compute 执行操作。

下面我们详细讨论每一个步骤。

### 1. 向 nova-api 发送请求

客户（可以是 OpenStack 最终用户，也可以是其他程序）向 API（nova-api）发送请求：“帮我暂停这个 Instance”，如图 7-71 所示。

```

View Log
Pause Instance
Suspend Instance
  
```

图 7-71

查看日志 `/opt/stack/logs/n-api.log`，如图 7-72 所示。

```

2016-01-08 18:33:58.484 DEBUG nova.api.openstack.wsgi [req-d0ccb9
a9-4e4f-4112-97e9-f61a99e3909a admin admin] Action: "action", call
ling method: <bound method PauseServerController._pause of <nova.
api.openstack.compute.pause_server.PauseServerController object a
t 0x7f151cd40a10>>, body: {"pause": null} _process_stack /opt/stack
/nova/nova/api/openstack/wsgi.py:789
  
```

图 7-72

注：对于 Pause 操作，日志没有前面 Start 记录得那么详细。

例如这里就没有记录 `nova.api.openstack.compute.servers` 和 `nova.compute.api` 代码模块的日志，这可能是因为这个操作逻辑比较简单，开发人员在编码时没有加入日志。

2. nova-api 发送消息

nova-api 向 Messaging (RabbitMQ) 发送了一条消息：“暂停这个 Instance”。查看源代码 /opt/stack/nova/nova/compute/api.py, 方法是 pause, 如图 7-73 所示。

```
2784 def pause(self, context, instance):
2785
2786     instance.task_state = task_states.PAUSING
2787     instance.save(expected_task_state=None)
2788     self.record_action_start(context, instance, instance_actions.PAUSE)
2789     self.compute_rpcapi.pause_instance(context, instance)
2790
```

图 7-73

3. nova-compute 执行操作

查看日志 /opt/stack/logs/n-cpu.log, 如图 7-74 所示。

```
2016-01-08 18:33:58.629 INFO nova.compute.manager [req-d0ccb9a9-4
e4f-4112-97e9-f61a99e3909a admin:admin] [instance: f1e22596-6844-
4d7a-84a3-e41e6d7618ef] Pausing
2016-01-08 18:33:58.631 DEBUG nova.objects.instance [req-d0ccb9a9
-4e4f-4112-97e9-f61a99e3909a admin:admin] Lazy-loading flavor: d
n Instance uuid f1e22596-6844-4d7a-84a3-e41e6d7618ef obj_load_att
r /opt/stack/nova/nova/objects/instance.py:860
2016-01-08 18:33:58.700 DEBUG nova.virt.driver [req-c787cd00-d5cd
-4d85-9a26-b247dcb6f90c None:None] Emitting event <LifecycleEvent
: 1452249238.7, f1e22596-6844-4d7a-84a3-e41e6d7618ef => Paused> e
mit_event /opt/stack/nova/nova/virt/driver.py:1303
2016-01-08 18:33:58.701 INFO nova.compute.manager [req-c787cd00-d
5cd-4d85-9a26-b247dcb6f90c None:None] [instance: f1e22596-6844-4d
7a-84a3-e41e6d7618ef] VM Paused (Lifecycle Event)
```

图 7-74

暂停操作成功执行后, instance 的状态变为 Paused, 如图 7-75 所示。

Status	Availability Zone	Task	Power State
Paused	nova	None	Paused

图 7-75

Resume 操作的日志分析留给大家练习。  
提示: 这里的 Resume 操作实际上是 Unpause 操作, 可以通过关键字 “unpause” 定位日志。

7.3.9 Suspend/Resume

有时需要长时间暂停 instance, 可以通过 Suspend 操作将 instance 的状态保存到宿主机的磁盘中。当需要恢复的时候, 执行 Resume 操作, 从磁盘读回 instance 的状态, 使之继续运行。这里需要对 Suspend 和 Pause 操作做个比较:

1. 相同点

两者都是暂停 instance 的运行, 并保存当前状态, 之后可以通过 Resume 操作恢复。



## 2. 不同点

(1) Suspend 将 instance 的状态保存在磁盘上；Pause 是保存在内存中，所以 Resume 被 Pause 的 instance 要比 Suspend 快。

(2) instance 被 Suspend 后，其状态是 Shut Down；而被 Pause 的 instance 状态是 Paused。

(3) 虽然都是通过 Resume 操作恢复，Pause 对应的 Resume 在 OpenStack 内部被叫作“Unpause”；Suspend 对应的 Resume 才是真正的“Resume”。这个在日志中能体现出来。

Suspend/Resume 的日志分析留给大家做练习。

### 7.3.10 Rescue/Unrescue

从这节开始，我们将讨论几种 instance 故障恢复的方法，不同方法适用于不同的场景。

首先我们考虑操作系统故障。有时候由于误操作或者突然断电，操作系统重启后却起不来了。为了最大限度挽救数据，我们通常会使用一张系统盘将系统引导起来，然后再尝试恢复。问题如果不太严重，完全可以通过这种方式让系统重新正常工作。比如，某个系统文件意外删除，root 密码遗忘等。Nova 也提供了这种故障恢复机制，叫做 Rescue。

我们来看看 rescue 的说明，如图 7-76 所示。

```
root@devstack-controller:~# nova help rescue
usage: nova rescue [--password <password>] [--image <image>] <server>

Reboots a server into rescue mode, which starts the machine from either the
initial image or a specified image, attaching the current boot disk as
secondary.

Positional arguments:
  <server>                Name or ID of server.

Optional arguments:
  --password <password>  The admin password to be set in the rescue
                           environment.
  --image <image>         The image to rescue with.
```

图 7-76

Rescue 用指定的 image 作为启动盘引导 instance，将 instance 本身的系统盘作为第二个磁盘挂载到操作系统上。

如图 7-77 所示是 rescue instance 的流程图。

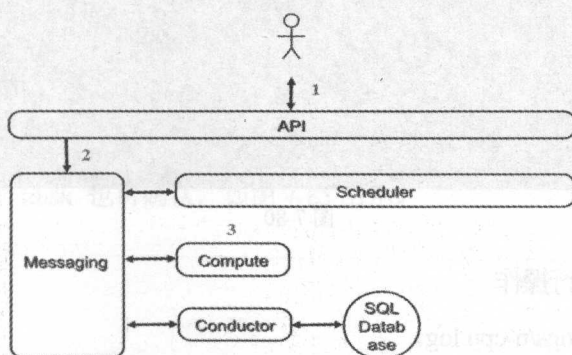


图 7-77

- 向 nova-api 发送请求。
- nova-api 发送消息。
- nova-compute 执行操作。

下面我们详细讨论每一个步骤。

### 1. 向 nova-api 发送请求

目前 Rescue 操作只能通过 CLI 执行，如图 7-78 所示。

```
root@devstack-controller:~# nova rescue c2
+-----+-----+
| Property | Value |
+-----+-----+
| adminPass | LKYZwzcTC94m |
+-----+-----+
```

图 7-78

这里我们没有指明用哪个 image 作为引导盘，nova 将使用 instance 部署时使用的 image 查看日志 /opt/stack/logs/n-api.log，如图 7-79 所示。

```
2016-01-19 19:00:23.347 DEBUG nova.api.openstack.wsgi [req-323ab557-a3ab-425
9-828a-f29a69d38336 admin admin] Action: 'action', calling method: <bound me
thod RescueController._rescue of <nova.api.openstack.compute.rescue.RescueCo
ntroller object at 0x7f151cd2dc0>.>, body: {'rescue': null} _process_stack /
opt/stack/nova/nova/api/openstack/wsgi.py:789
```

图 7-79

### 2. nova-api 发送消息

nova-api 向 Messaging (RabbitMQ) 发送了一条消息：“Rescue 这个 Instance” 查看源代码/opt/stack/nova/nova/compute/api.py，方法是 rescue，如图 7-80 所示。

```
2839 def rescue(self, context, instance, rescue_password=None,
2840           rescue_image_ref=None, clean_shutdown=True):
2841
2842
2843     bdms = objects.BlockDeviceMappingList.get_by_instance_uuid(
2844         context, instance.uuid)
2845     for bdm in bdms:
2846         if bdm.volume_id:
2847             vol = self.volume_api.get(context, bdm.volume_id)
2848             self.volume_api.check_attached(context, vol)
2849     if self.is_volume_backed_instance(context, instance, bdms):
2850         reason = _
2851         raise exception.InstanceNotRescuable(instance_id=instance.uuid,
2852                                               reason=reason)
2853
2854     instance.task_state = task_states.RESCUING
2855     instance.save(expected_task_state=[None])
2856
2857     self._record_action_start(context, instance, instance.actions.RESCUE)
2858
2859     self.compute_rpcapi.rescue_instance(context, instance=instance,
2860                                         rescue_password=rescue_password, rescue_image_ref=rescue_image_ref,
2861                                         clean_shutdown=clean_shutdown)
```

图 7-80

### 3. nova-compute 执行操作

查看日志 /opt/stack/logs/n-cpu.log。

(1) 关闭 instance，如图 7-81 所示。

```
2016-01-19 19:00:24.014 INFO nova.compute.manager [req-323ab557-a3ab-4259-828a-f29a69d38336 admin admin] [instance: 3aebf982-09f0-4b6a-b763-9ce594277963] Rescuing
2016-01-19 19:00:24.556 DEBUG nova.virt.libvirt.driver [req-323ab557-a3ab-4259-828a-f29a69d38336 admin admin] [instance: 3aebf982-09f0-4b6a-b763-9ce594277963] Shutting down instance from state 1_clean_shutdown /opt/stack/nova/nova/virt/libvirt/driver.py:2227
2016-01-19 19:00:27.580 INFO nova.virt.libvirt.driver [req-323ab557-a3ab-4259-828a-f29a69d38336 admin admin] [instance: 3aebf982-09f0-4b6a-b763-9ce594277963] Instance shutdown successfully after 3 seconds.
```

图 7-81

(2) 通过 image 创建新的引导盘，命名为 disk.rescue，如图 7-82 所示。

```
2016-01-19 19:00:27.713 DEBUG oslo_concurrency.processutils [req-323ab557-a3ab-4259-828a-f29a69d38336 admin admin] Running cmd (subprocess): qemu-img create -f qcow2 -o backing_file=/opt/stack/data/nova/instances/_base/60bba5916c6c90ed2ef7d3263de8f653111dd35f /opt/stack/data/nova/instances/3aebf982-09f0-4b6a-b763-9ce594277963/disk.rescue execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
```

图 7-82

(3) 启动 instance，如图 7-83 所示。

```
2016-01-19 19:00:28.040 DEBUG nova.virt.libvirt.driver [req-323ab557-a3ab-4259-828a-f29a69d38336 admin admin] [instance: 3aebf982-09f0-4b6a-b763-9ce594277963] Start _get_guest_xml network_info=[VIF({'profile': {}, 'ovs_interface_id': None, 'preserve_on_delete': False, 'network': Network({'bridge': 'brq5333bf1-67', 'subnets': [Subnet({'ips': [FixedIP({'version': 4, 'vif_mac': 'u
```

图 7-83

Rescue 执行成功后，可以通过 `virsh edit <instance>` 查看 instance 的 XML 定义，disk.rescue 作为启动盘 vda，真正的启动盘 disk 作为第二个磁盘 vdb，如图 7-84 所示。

```
<disk type='file' device='disk'>
  <driver name='qemu' type='qcow2' cache='on'>
    <source file='/opt/stack/data/nova/instances/3aebf982-09f0-4b6a-b763-9ce594277963/disk.rescue'>
    <target dev='vda' bus='virtio'>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x00' function='0x0'>
  </disk>
  <disk type='file' device='disk'>
    <driver name='qemu' type='qcow2' cache='on'>
    <source file='/opt/stack/data/nova/instances/3aebf982-09f0-4b6a-b763-9ce594277963/disk'>
    <target dev='vdb' bus='virtio'>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x00' function='0x0'>
  </disk>
```

图 7-84

登录 instance，通过 `fdisk` 也可确认，如图 7-85 所示。

```
$ sudo fdisk -l

Disk /dev/vda: 41 MB, 41126400 bytes
255 heads, 63 sectors/track, 5 cylinders, total 80325 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

   Device Boot      Start         End      Blocks    Id  System
/dev/vda1 *          16065         80324       32130    83  Linux

Disk /dev/vdb: 41 MB, 41126400 bytes
255 heads, 63 sectors/track, 5 cylinders, total 80325 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

   Device Boot      Start         End      Blocks    Id  System
/dev/vdb1 *          16065         80324       32130    83  Linux
```

图 7-85

此时，instance 处于 Rescue 状态，如图 7-86 所示。

<input type="checkbox"/>	Instance Name	Image Name	IP Address	Size	Key Pair	Status
<input type="checkbox"/>	c2	cirros	10.10.1.20	m1.nano	-	Rescue

图 7-86

Rescue 操作让我们有机会修复损坏的操作系统。

修好之后，使用 Unrescue 操作从原启动盘重新引导 instance，如图 7-87 所示。

```
usage: nova unrescue <server>
Restart the server from normal boot disk again.

Positional arguments:
  <server> Name or ID of server.
root@devstack-controller:~# nova unrescue c2
root@devstack-controller:~#
```

图 7-87

Unrescue 的日志分析留给大家练习。

7.3.11 Snapshot

有时候操作系统损坏得很严重，通过 Rescue 操作无法修复，那么我们就得考虑通过备份恢复了。当然前提是我们之前对 instance 做过备份。

Nova 备份的操作叫 Snapshot，其工作原理是对 instance 的镜像文件（系统盘）进行全量备份，生成一个类型为 snapshot 的 image，然后将其保存到 Glance 上。

从备份恢复的操作叫 Rebuild，将在下一节重点讨论。

如图 7-88 所示是 snapshot instance 的流程图。



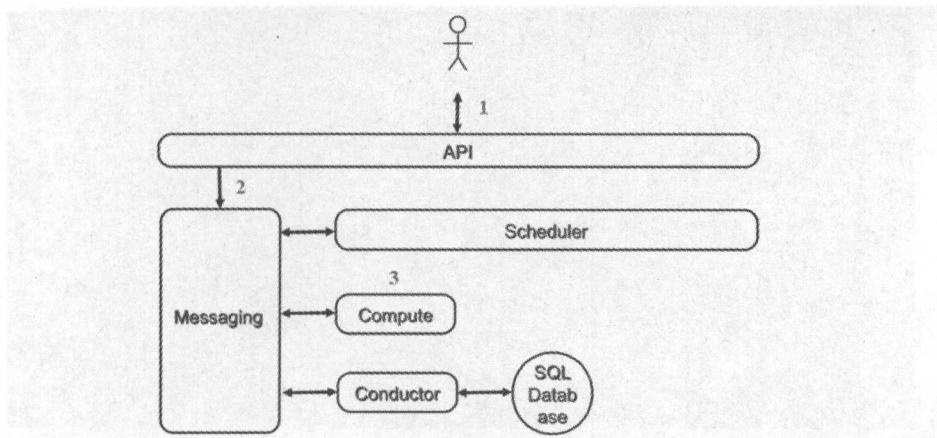


图 7-88

- 向 nova-api 发送请求。
- nova-api 发送消息。
- nova-compute 执行操作。

下面我们详细讨论每一个步骤。

### 1. 向 nova-api 发送请求

客户（可以是 OpenStack 最终用户，也可以是其他程序）向 API（nova-api）发送请求：“对这个 Instance 做个快照”，如图 7-89 所示。

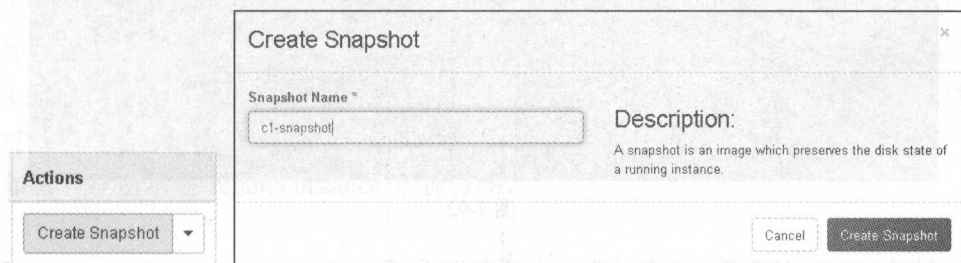


图 7-89

查看日志 `/opt/stack/logs/n-api.log`，如图 7-90 所示。

```

2016-01-19 17:48:09.005 DEBUG nova.api.openstack.wsgi [req-facc29bd-0c4d-4b2
6-b13f-fcae07608969 admin admin] Action: 'action', calling method: <bound me
thod ServersController._action_create_image of <nova.api.openstack.compute.s
ervers.ServersController object at 0x7f151ce6ef90>, body: {"createImage": {
"name": "c1-snapshot", "metadata": {}}} _process_stack /opt/stack/nova/nova/
api/openstack/wsgi.py:789
  
```

图 7-90

### 2. nova-api 发送消息

nova-api 向 Messaging（RabbitMQ）发送了一条消息：“对这个 Instance 做快照”。查看源代码 `/opt/stack/nova/nova/compute/api.py`，方法是 `snapshot`，如图 7-91 所示。

```

2239     def snapshot(self, context, instance, name, extra_properties=None):
2240
2241
2242
2243
2244
2245
2246
2247
2248         image_meta = self._create_image(context, instance, name,
2249                                         extra_properties=extra_properties)
2250
2251
2252         NOTE
2253
2254         instance.task_state = task_states.IMAGE_SNAPSHOT_PENDING
2255         instance.save(expected_task_state=[None])
2256
2257         self.compute_rpcapi.snapshot_instance(context, instance,
2258                                                image_meta[ ])
2259
2260     return image_meta

```

图 7-91

### 3. nova-compute 执行操作

查看日志 /opt/stack/logs/n-cpu.log。

- 暂停 instance，如图 7-92 所示。

```

2016-01-19 17:48:09.485 INFO nova.compute.manager [req-facc29bd-0c4d-4b26-b13f-fcae07608969 admin admin] [instance: 0229842d-5872-4c77-a790-e3573e7abeca] instance_snapshotting
2016-01-19 17:48:09.605 DEBUG oslo_concurrency.processutils [req-facc29bd-0c4d-4b26-b13f-fcae07608969 admin admin] Running cmd (subprocess): env LC_ALL=C LANG=C qemu-img info /opt/stack/data/nova/instances/0229842d-5872-4c77-a790-e3573e7abeca/disk execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-01-19 17:48:09.632 DEBUG oslo_concurrency.processutils [req-facc29bd-0c4d-4b26-b13f-fcae07608969 admin admin] CMD "env LC_ALL=C LANG=C qemu-img info /opt/stack/data/nova/instances/0229842d-5872-4c77-a790-e3573e7abeca/disk" returned: 0 in 0.027s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
2016-01-19 17:48:09.651 DEBUG nova.virt.driver [req-e3f82c49-be12-4a4d-8a6a-6dd39dcdc7bc None None] Emitting event <LifecycleEvent: 1453196889.65, 0229842d-5872-4c77-a790-e3573e7abeca => Paused> emit_event /opt/stack/nova/nova/virt/driver.py:1303
2016-01-19 17:48:09.652 INFO nova.compute.manager [req-e3f82c49-be12-4a4d-8a6a-6dd39dcdc7bc None None] [instance: 0229842d-5872-4c77-a790-e3573e7abeca] VM_Paused (Lifecycle Event)

```

图 7-92

- 对 instance 的镜像文件做快照，如图 7-93 所示。

```

2016-01-19 17:48:10.674 INFO nova.virt.libvirt.driver [req-facc29bd-0c4d-4b26-b13f-fcae07608969 admin admin] [instance: 0229842d-5872-4c77-a790-e3573e7abeca] Beginning cold snapshot process
2016-01-19 17:48:10.757 DEBUG oslo_concurrency.processutils [req-facc29bd-0c4d-4b26-b13f-fcae07608969 admin admin] Running cmd (subprocess): qemu-img convert -f qcow2 -O qcow2 /opt/stack/data/nova/instances/0229842d-5872-4c77-a790-e3573e7abeca/disk /opt/stack/data/nova/instances/snapshots/tmpGI9NX0/5ba2f8cealb04c618c1e54d6f55ae6d5 execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-01-19 17:48:10.943 DEBUG oslo_concurrency.processutils [req-facc29bd-0c4d-4b26-b13f-fcae07608969 admin admin] CMD "qemu-img convert -f qcow2 -O qcow2 /opt/stack/data/nova/instances/0229842d-5872-4c77-a790-e3573e7abeca/disk /opt/stack/data/nova/instances/snapshots/tmpGI9NX0/5ba2f8cealb04c618c1e54d6f55ae6d5" returned: 0 in 0.185s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280

```

图 7-93

- 恢复 instance，如图 7-94 所示。

```

2016-01-19 17:48:11.860 INFO nova.compute.manager [req-e3f82c49-be12-4a4d-8a
6a-6dd39dcdc7bc None None] [instance: 0229842d-5872-4c77-a790-e3573e7abeca]
VM Started (Lifecycle Event)
2016-01-19 17:48:12.217 INFO nova.compute.manager [req-e3f82c49-be12-4a4d-8a
6a-6dd39dcdc7bc None None] [instance: 0229842d-5872-4c77-a790-e3573e7abeca]
VM Resumed (Lifecycle Event)

```

图 7-94

- 将快照上传到 Glance，如图 7-95 所示。

```

2016-01-19 17:48:12.157 INFO nova.virt.libvirt.driver [req-facc29bd-0c4d-4b2
6-b13f-fcae07608969 admin admin] [instance: 0229842d-5872-4c77-a790-e3573e7a
beca] Snapshot extracted, beginning image upload
2016-01-19 17:48:12.715 INFO nova.virt.libvirt.driver [req-facc29bd-0c4d-4b2
6-b13f-fcae07608969 admin admin] [instance: 0229842d-5872-4c77-a790-e3573e7a
beca] Snapshot image upload complete

```

图 7-95

- Snapshot 成功保存在 Glance 中，如图 7-96 所示。

<input type="checkbox"/>	Image Name	Type
<input type="checkbox"/>	c1-snapshot	Snapshot

图 7-96

### 7.3.12 Rebuild

上一节我们讨论了 snapshot，snapshot 的一个重要作用是对 instance 做备份。

如果 instance 损坏了，可以通过 snapshot 恢复，这个恢复的操作就是 Rebuild。

Rebuild 会用 snapshot 替换 instance 当前的镜像文件，同时保持 instance 的其他，诸如网络，资源分配属性不变。

如图 7-97 所示是 rebuild instance 的流程图。

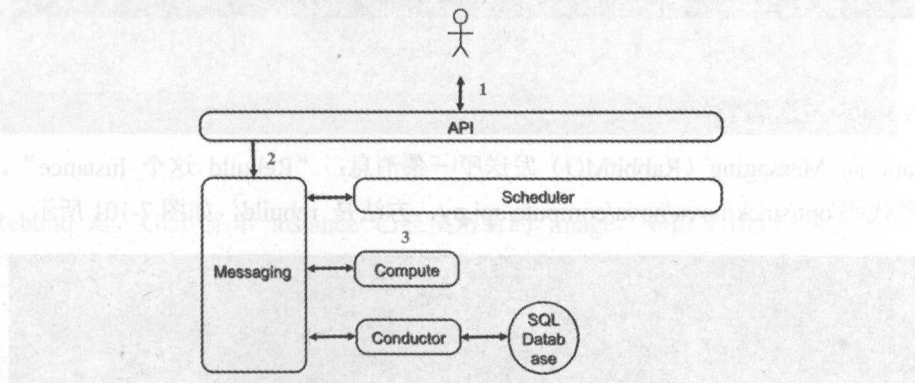


图 7-97

- 向 nova-api 发送请求。

- nova-api 发送消息。
- nova-compute 执行操作。

下面我们详细讨论每一个步骤。

### 1. 向 nova-api 发送请求

客户（可以是 OpenStack 最终用户，也可以是其他程序）向 API（nova-api）发送请求：“Rebuild 这个 Instance”，如图 7-98 所示。

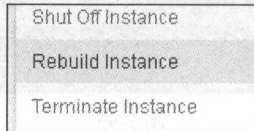


图 7-98

选择用于恢复的 image，如图 7-99 所示。

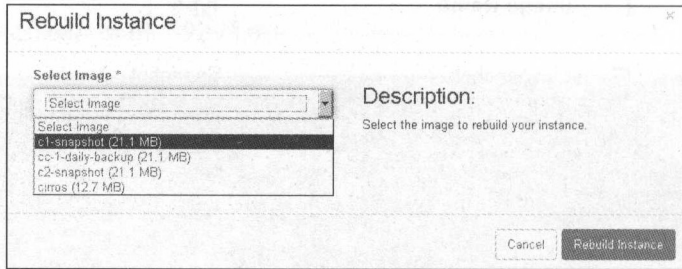


图 7-99

查看日志/opt/stack/logs/n-api.log，如图 7-100 所示。

```
2016-01-19 18:18:38.859 DEBUG:nova.api.openstack.wsgi [req-29291a6f-c17b-4c7
1-b4e8-edb2349f0b7e admin admin] Action: 'action', calling method: <bound me
thod ServersController._action_rebuild of <nova.api.openstack.compute.server
s.ServersController object at 0x7f151ce6ef90>, body: {"rebuild": {"OS-DCF:dis
kconfig": "AUTO", "imageRef": "b16a6785-616a-4dec-909f-b2a0e3063f36"}}} _pr
ocess_stack /opt/stack/nova/nova/api/openstack/wsgi.py:789
```

图 7-100

### 2. nova-api 发送消息

nova-api 向 Messaging（RabbitMQ）发送了一条消息：“Rebuild 这个 Instance”。查看源代码/opt/stack/nova/nova/compute/api.py，方法是 rebuild，如图 7-101 所示。

```
2511         self.compute_task_api.rebuild_instance(context, instance=instance,
2512         new_pass=admin_password, injected_files=files_to_inject,
2513         image_ref=image_href, orig_image_ref=orig_image_ref,
2514         orig_sys_metadata=orig_sys_metadata, bdms=bdms,
2515         preserve_ephemeral=preserve_ephemeral, host=instance.host,
2516         kwargs=kwargs)
2517
```

图 7-101



3. nova-compute 执行操作

查看日志 /opt/stack/logs/n-cpu.log。

- 关闭 instance，如图 7-102 所示。

```
2016-01-19 18:18:39.218 INFO nova.compute.manager [req-29291a6f-c17b-4c71-b4e8-edb2349f0b7e admin admin] [instance: 0229842d-5872-4c77-a790-e3573e7abeca] Rebuilding instance
2016-01-19 18:18:39.543 DEBUG nova.virt.libvirt.driver [req-29291a6f-c17b-4c71-b4e8-edb2349f0b7e admin admin] [instance: 0229842d-5872-4c77-a790-e3573e7abeca] Shutting down instance from state 1 _clean_shutdown /opt/stack/nova/nova/virt/libvirt/driver.py:2227
2016-01-19 18:18:41.966 DEBUG oslo.service.periodic_task [req-dba74954-766b-4c74-b394-cfa5461643d3 None None] Running periodic task ComputeManager._poll_rebooting_instances_run_periodic_tasks /usr/local/lib/python2.7/dist-packages/oslo_service/periodic_task.py:213
2016-01-19 18:18:42.574 INFO nova.virt.libvirt.driver [req-29291a6f-c17b-4c71-b4e8-edb2349f0b7e admin admin] [instance: 0229842d-5872-4c77-a790-e3573e7abeca] Instance shutdown successfully after 3 seconds.
```

图 7-102

- 下载新的 image，并准备 instance 的镜像文件，如图 7-103 所示。

```
2016-01-19 18:18:42.962 INFO nova.virt.libvirt.driver [req-29291a6f-c17b-4c71-b4e8-edb2349f0b7e admin admin] [instance: 0229842d-5872-4c77-a790-e3573e7abeca] Creating image
2016-01-19 18:18:43.432 DEBUG oslo_concurrency.processutils [req-29291a6f-c17b-4c71-b4e8-edb2349f0b7e admin admin] Running cmd (subprocess): qemu-img create -f qcow2 -o backing_file=/opt/stack/data/nova/instances/_base/4876/daa14c0481350666fbca4a015cba2933716 /opt/stack/data/nova/instances/0229842d-5872-4c77-a790-e3573e7abeca/disk execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
```

图 7-103

- 启动 instance，如图 7-104 所示。

```
2016-01-19 18:18:43.659 DEBUG nova.virt.libvirt.driver [req-29291a6f-c17b-4c71-b4e8-edb2349f0b7e admin admin] [instance: 0229842d-5872-4c77-a790-e3573e7abeca] Start _get_guest_xml network_info=[VIF({'profile': {}, 'ovs_interface_id': None, 'preserve_on_delete': False, 'network': Network({'bridge': 'brq
2016-01-19 18:18:44.356 INFO nova.compute.manager [req-e3f82c49-be12-4a4d-8a6a-6dd39dcdc7bc None None] [instance: 0229842d-5872-4c77-a790-e3573e7abeca] VM Started (Lifecycle Event)
2016-01-19 18:18:44.359 DEBUG nova.virt.libvirt.driver [req-29291a6f-c17b-4c71-b4e8-edb2349f0b7e admin admin] [instance: 0229842d-5872-4c77-a790-e3573e7abeca] Instance is running spawn /opt/stack/nova/nova/virt/libvirt/driver.py:2445
2016-01-19 18:18:44.369 2710 INFO nova.virt.libvirt.driver [-] [instance: 0229842d-5872-4c77-a790-e3573e7abeca] Instance spawned successfully.
```

图 7-104

- Rebuild 后，GUI 显示 instance 已经使用新的 image，如图 7-105 所示。

<input type="checkbox"/>	Instance Name	Image Name
<input type="checkbox"/>	c1	c1-snapshot

图 7-105

### 7.3.13 Shelve

Instance 被 Suspend 后虽然处于 Shut Down 状态, 但 Hypervisor 依然在宿主机上为其预留了资源, 以便在以后能够成功 Resume。

如果希望释放这些预留资源，可以使用 `Shelve` 操作。

Shelve 会将 instance 作为 image 保存到 Glance 中，然后在宿主机上删除该 instance。

如图 7-106 所示是 shelve instance 的流程图。

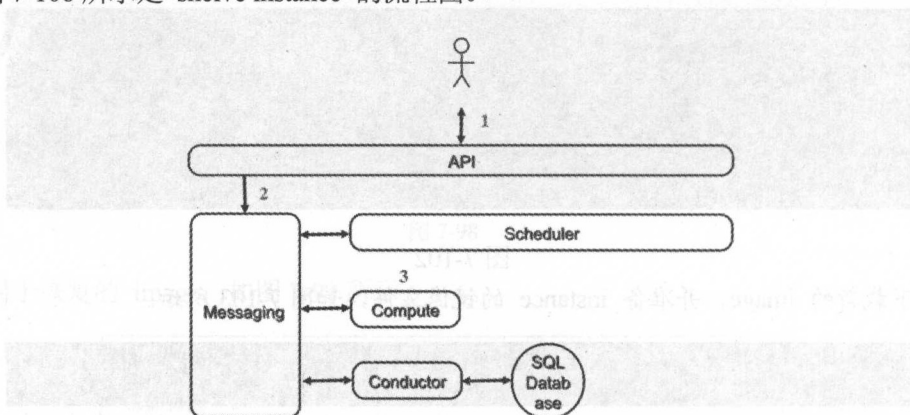


图 7-106

- 向 nova-api 发送请求。
- nova-api 发送消息。
- nova-compute 执行操作。

下面我们详细讨论每一个步骤。

## 1. 向 nova-api 发送请求

客户（可以是 OpenStack 最终用户，也可以是其他程序）向 API (nova-api) 发送请求：“帮我 shelve 这个 Instance”，如图 7-107 所示。

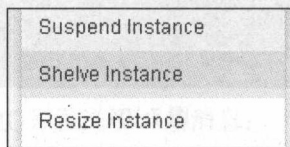


图 7-107

查看日志 `/opt/stack/logs/n-api.log`，如图 7-108 所示。

```
2016-01-08 23:06:59.562 DEBUG nova.api.openstack.wsgi [req-4985a5
6f-00d7-452e-90b3-d657565c3c83 admin admin] Action: 'action', call
ing method: <bound method ShelveController._shelve of <nova.api
openstack.compute.shelve.ShelveController object at 0x7f151cccc0
a0>., body: {'shelve': null} _process_stack /opt/stack/nova/nova/a
pi/openstack/wsgi.py:789
```

图 7-108

## 2. nova-api 发送消息

nova-api 向 Messaging (RabbitMQ) 发送了一条消息：“shelve 这个 Instance”  
查看源代码 /opt/stack/nova/nova/compute/api.py, 方法是 shelve, 如图 7-109 所示。

```
def shelve(self, context, instance, clean_shutdown=True):

    instance.task_state = task_states.SHELVING
    instance.save(expected_task_state=[None])

    self._record_action_start(context, instance, instance_actions.SHELVING)

    if not self.is_volume_backed_instance(context, instance):
        name = % instance.display_name
        image_meta = self._create_image(context, instance, name,
            image_id = image_meta['id'])
        self.compute_rpcapi.shelve_instance(context, instance=instance,
            image_id=image_id, clean_shutdown=clean_shutdown)
```

图 7-109

## 3. nova-compute 执行操作

查看日志 /opt/stack/logs/n-cpu.log。

- 首先, 关闭 instance, 如图 7-110 所示。

```
2016-01-08 23:07:00.045 DEBUG nova.virt.libvirt.driver [req-4985a96f-00d7-452e-90b3-d657565c3c83 admin admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] Shutting down instance from state 1
2016-01-08 23:07:03.234 INFO nova.virt.libvirt.driver [req-4985a96f-00d7-452e-90b3-d657565c3c83 admin admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] Instance shutdown successfully after 3 seconds.
```

图 7-110

然后对 instance 执行 snapshot 操作, 如图 7-111 所示。

```
2016-01-08 23:07:03.144 INFO nova.virt.libvirt.driver [req-4985a96f-00d7-452e-90b3-d657565c3c83 admin admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] Beginning cold snapshot process
2016-01-08 23:07:03.234 DEBUG oslo_concurrency.processutils [req-4985a96f-00d7-452e-90b3-d657565c3c83 admin admin] Running cmd (subprocess): qemu-img convert -f qcow2 -o qcow2 /opt/stack/data/nova/instances/f1e22596-6844-4d7a-84a3-e41e6d7618ef/disk /opt/stack/data/nova/instances/snapshots/tmpNyFLBj/4fc7b4700c3047d7ab5172e28eb33920.execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-01-08 23:07:03.386 DEBUG oslo_concurrency.processutils [req-4985a96f-00d7-452e-90b3-d657565c3c83 admin admin] CMD "qemu-img convert -f qcow2 -o qcow2 /opt/stack/data/nova/instances/f1e22596-6844-4d7a-84a3-e41e6d7618ef/disk /opt/stack/data/nova/instances/snapshots/tmpNyFLBj/4fc7b4700c3047d7ab5172e28eb33920" returned: 0
2016-01-08 23:07:03.387 INFO nova.virt.libvirt.driver [req-4985a96f-00d7-452e-90b3-d657565c3c83 admin admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] Snapshot extracted, beginning image upload
2016-01-08 23:07:04.026 INFO nova.virt.libvirt.driver [req-4985a96f-00d7-452e-90b3-d657565c3c83 admin admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] Snapshot image upload complete
2016-01-08 23:07:04.031 DEBUG nova.compute.manager [req-4985a96f-00d7-452e-90b3-d657565c3c83 admin admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] checking state _get_power_state /opt/stack/nova/nova/compute/manager.py:1331
```

图 7-111

- 成功后，snapshot 生成的 image 会保存在 Glance 上，命名为 <instance name>-shelved，如图 7-112 所示。

Image Name	Type
c2-shelved	Snapshot

图 7-112

- 最后删除 instance 在宿主机上的资源，如图 7-113 所示。

```
66f-00d7-452e-90b3-d657565c3c83 admin admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] Deleting instance files /opt/stack/data/nova/instances/f1e22596-6844-4d7a-84a3-e41e6d7618ef_delete
2016-01-08 23:07:04.391 INFO nova.virt.libvirt.driver [req-4985a9-66f-00d7-452e-90b3-d657565c3c83 admin admin] [instance: f1e22596-6844-4d7a-84a3-e41e6d7618ef] Deletion of /opt/stack/data/nova/instances/f1e22596-6844-4d7a-84a3-e41e6d7618ef_delete complete
```

图 7-113

暂停操作成功执行后，instance 的状态变为 Shelved Offloaded，电源状态是 Shut Down，如图 7-114 所示。

Status	Availability Zone	Task	Power State
Shelved Offloaded	nova	None	Shut Down

图 7-114

### 7.3.14 Unshelve

通过 Unshelve 操作恢复被 shelve 的 instance。

因为 Glance 中保存了 instance 的 image，unshelve 的过程其实就是通过该 image launch 一个新的 instance，nova-scheduler 也会调度合适的计算节点来创建该 instance。

instance unshelve 后可能运行在与 shelve 之前不同的计算节点上，但 instance 的其他属性（比如 flavor，IP 等）不会改变。

如图 7-115 所示是 Unshelve instance 的流程图。



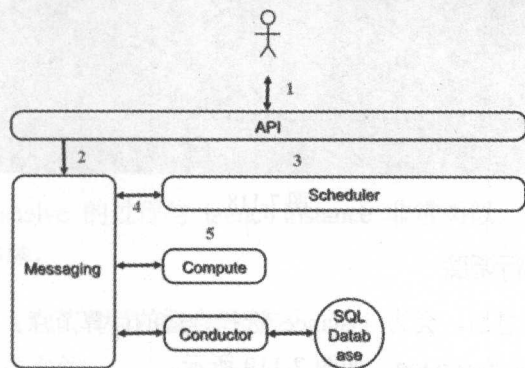


图 7-115

- 向 nova-api 发送请求。
- nova-api 发送消息。
- nova-scheduler 执行调度。
- nova-scheduler 发送消息。
- nova-compute 执行操作。

下面我们详细讨论每一个步骤。

### 1. 向 nova-api 发送请求

客户（可以是 OpenStack 最终用户，也可以是其他程序）向 API（nova-api）发送请求：“帮我 Unshelve 这个 Instance”，如图 7-116 所示。

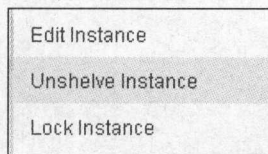


图 7-116

查看日志 /opt/stack/logs/n-api.log，如图 7-117 所示。

```

2016-01-10 18:38:36.223 DEBUG nova.api.openstack.wsgi [req-b6658e
7c-9df6-40ae-ace2-78f4e48d5792 admin:admin] Action: 'action', ca
ling method: <bound method ShelveController._unshelve of <nova.ap
i.openstack.compute.shelve.ShelveController object at 0x7f151cccl
050>X, body: {'unshelve': null} _process_stack /opt/stack/nova/no
va/api/openstack/wsgi.py:789
  
```

图 7-117

### 2. nova-api 发送消息

nova-api 向 Messaging（RabbitMQ）发送了一条消息：“unshelve 这个 Instance”

查看源代码 /opt/stack/nova/nova/compute/api.py，方法是 unshelve，如图 7-118 所示。

```

def unselve(self, context, instance):
    instance.task_state = task_states.UNSHELVING
    instance.save(expected_task_state=[None])

    self._record_action_start(context, instance, instance_actions.UNSHELVE)

    self.compute_task_api.unshelve_instance(context, instance)

```

图 7-118

### 3. nova-scheduler 执行调度

nova-scheduler 收到消息后, 会为 instance 选择合适的计算节点。

查看日志 /opt/stack/logs/n-sch.log, 如图 7-119 所示。

```

2016-01-10 18:38:36.745 DEBUG nova.filters [req-b6658e7c-9df6-40a
e-ace2-78f4e48d5792 admin admin] Starting with 2 host(s) get_filt
ered_objects /opt/stack/nova/nova/filters.py:70
2016-01-10 18:38:36.745 DEBUG nova.filters [req-b6658e7c-9df6-40a
e-ace2-78f4e48d5792 admin admin] Filter RetryFilter returned 2 ho
st(s) get_filtered_objects /opt/stack/nova/nova/filters.py:104
2016-01-10 18:38:36.746 DEBUG nova.filters [req-b6658e7c-9df6-40a
e-ace2-78f4e48d5792 admin admin] Filter AvailabilityZoneFilter re
turned 2 host(s) get_filtered_objects /opt/stack/nova/nova/filters
.py:104
2016-01-10 18:38:36.746 DEBUG nova.filters [req-b6658e7c-9df6-40a
e-ace2-78f4e48d5792 admin admin] Filter RamFilter returned 2 host
(s) get_filtered_objects /opt/stack/nova/nova/filters.py:104
2016-01-10 18:38:36.746 DEBUG nova.filters [req-b6658e7c-9df6-40a
e-ace2-78f4e48d5792 admin admin] Filter DiskFilter returned 2 ho
st(s) get_filtered_objects /opt/stack/nova/nova/filters.py:104
2016-01-10 18:38:36.748 DEBUG nova.filters [req-b6658e7c-9df6-40a
e-ace2-78f4e48d5792 admin admin] Filter ServerGroupAffinityFilter
returned 2 host(s) get_filtered_objects /opt/stack/nova/nova/filt
ers.py:104
2016-01-10 18:38:36.748 DEBUG nova.scheduler.filter_scheduler [re
q-b6658e7c-9df6-40ae-ace2-78f4e48d5792 admin admin] Filtered [(de
vstack-compute1, devstack-compute1) ram:9424 disk:4096 io_ops:0
instances:1, (devstack-controller, devstack-controller) ram:9488
disk:5120 io_ops:0 instances:0] _schedule /opt/stack/nova/nova/sch
eduler/filter_scheduler.py:152
2016-01-10 18:38:36.748 DEBUG nova.scheduler.filter_scheduler [re
q-b6658e7c-9df6-40ae-ace2-78f4e48d5792 admin admin] weighed [weig
hedHost [host: (devstack-controller, devstack-controller) ram:948
8 disk:5120 io_ops:0 instances:0, weight: 1.0], weighedHost [host
: (devstack-compute1, devstack-compute1) ram:9424 disk:4096 io_op
s:0 instances:1, weight: 0.993254637437]] _schedule /opt/stack/no
va/nova/scheduler/filter_scheduler.py:157
2016-01-10 18:38:36.749 DEBUG nova.scheduler.filter_scheduler [re
q-b6658e7c-9df6-40ae-ace2-78f4e48d5792 admin admin] Selected host
: weighedHost [host: (devstack-controller, devstack-controller) r
am:9488 disk:5120 io_ops:0 instances:0, weight: 1.0] _schedule /o
pt/stack/nova/nova/scheduler/filter_scheduler.py:167

```

图 7-119

经过筛选, 最终 devstack-controller 被选中 launch instance。

### 4. nova-scheduler 发送消息

nova-scheduler 发送消息, 告诉被选中的计算节点可以 launch instance 了。源代码在 /opt/stack/nova/nova/scheduler/filter\_scheduler.py 第 95 行, 方法为 select\_destinations, 如图 7-120 所示。

```

94
95     self.notifier.info(context,
96                        dict(request_spec=request_spec))

```

图 7-120

### 5. nova-compute 执行操作

nova-compute 执行 unshelve 的过程与 launch instance 非常类似。一样会经过如下几个步骤：

- (1) 为 instance 准备 CPU、内存和磁盘资源。
- (2) 创建 instance 镜像文件。
- (3) 创建 instance 的 XML 定义文件。
- (4) 创建虚拟网络并启动 instance。

日志记录在 /opt/stack/logs/n-cpu.log，分析留给大家练习。

## 7.3.15 Migrate

Migrate 操作的作用是将 instance 从当前的计算节点迁移到其他节点上。Migrate 不要求源和目标节点必须共享存储，当然共享存储也是可以的。Migrate 前必须满足一个条件：计算节点间需要配置 nova 用户无密码访问。

如图 7-121 所示是 Migrate instance 的流程图。

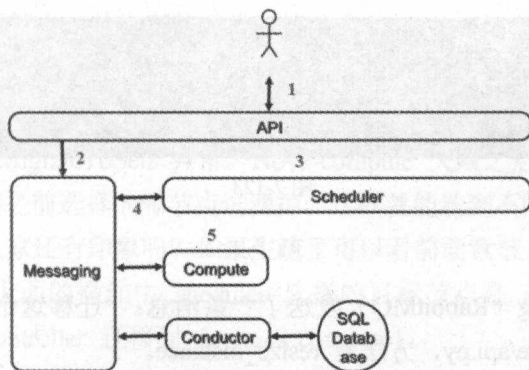


图 7-121

- 向 nova-api 发送请求。
- nova-api 发送消息。
- nova-scheduler 执行调度。
- nova-scheduler 发送消息。
- nova-compute 执行操作。

下面我们详细讨论每一个步骤。

1. 向 nova-api 发送请求

客户（可以是 OpenStack 最终用户，也可以是其他程序）向 API (nova-api) 发送请求：“帮我迁移这个 Instance”

Migrate 操作是特权操作，只能在 Admin 的 instance 菜单中执行，如图 7-122 所示。

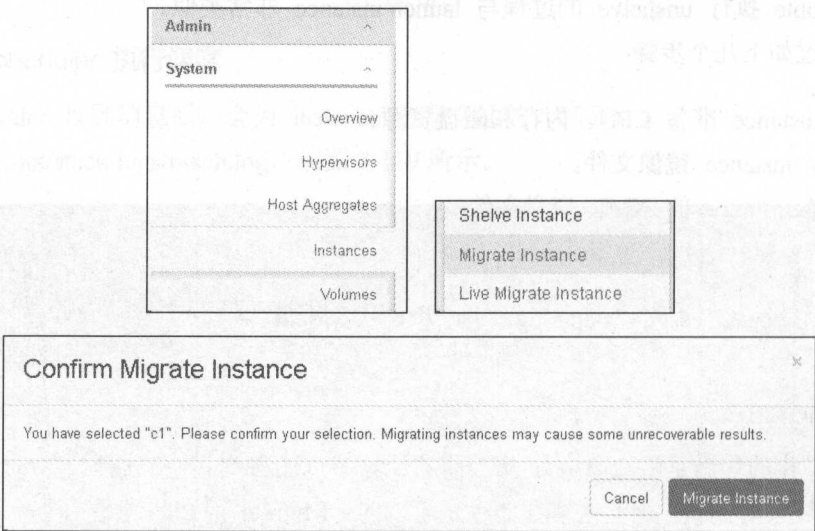


图 7-122

查看日志 /opt/stack/logs/n-api.log ，如图 7-123 所示。

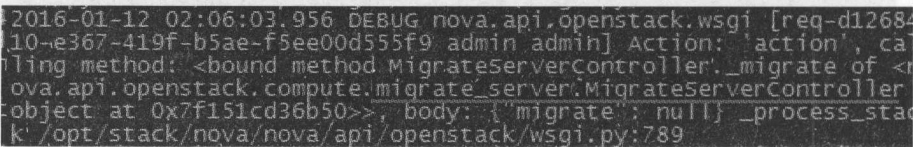


图 7-123

2. nova-api 发送消息

nova-api 向 Messaging (RabbitMQ) 发送了一条消息：“迁移这个 Instance”。查看源代码 /opt/stack/nova/nova/compute/api.py, 方法是 `resize_instance`。

- 没错，是 `resize` 而非 `migrate`。

这是由于 `migrate` 实际上是通过 `resize` 操作实现的，至于为什么要这样设计，我们会在下一节 `resize` 中详细分析，如图 7-124 所示。

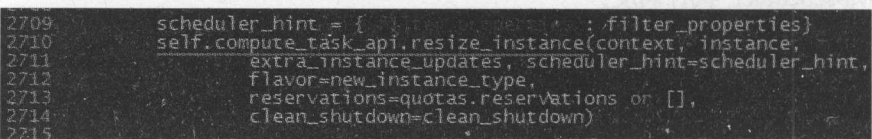


图 7-124



### 3. nova-scheduler 执行调度

nova-scheduler 收到消息后, 会为 instance 选择合适的目标计算节点。

查看日志 /opt/stack/logs/n-sch.log, 如图 7-125 所示。

```
2016-01-12 02:06:04.209 DEBUG nova.scheduler.filter_scheduler [req-d1268410-e367-419f-b5ae-f5ee00d555f9 admin admin] Filtered [(devstack-compute1, devstack-compute1) ram:9488 disk:5120 io_ops:0 instances:0, (devstack-controller, devstack-controller) ram:9360 disk:4096 io_ops:0 instances:1] _schedule /opt/stack/nova/nova/scheduler/filter_scheduler.py:152
2016-01-12 02:06:04.209 DEBUG nova.scheduler.filter_scheduler [req-d1268410-e367-419f-b5ae-f5ee00d555f9 admin admin] weighed [weighedHost [host: (devstack-compute1, devstack-compute1) ram:9488 disk:5120 io_ops:0 instances:0, weight: 1.0], weighedHost [host: (devstack-controller, devstack-controller) ram:9360 disk:4096 io_ops:0 instances:1, weight: 0.986509274874]] _schedule /opt/stack/nova/nova/scheduler/filter_scheduler.py:157
2016-01-12 02:06:04.209 DEBUG nova.scheduler.filter_scheduler [req-d1268410-e367-419f-b5ae-f5ee00d555f9 admin admin] selected host [0: weighedHost [host: (devstack-compute1, devstack-compute1) ram:9488 disk:5120 io_ops:0 instances:0, weight: 1.0]] _schedule /opt/stack/nova/nova/scheduler/filter_scheduler.py:167
```

图 7-125

可以看到, 因为 devstack-compute1 的权值比 devstack-controller 大, 最终选择 devstack-compute1 作为目标节点。

看到上面的日志, 大家发现问题没有?

在分析这段日志的时候, 我发现 scheduler 选出来的计算节点有可能是当前节点源节点! 因为 scheduler 并没在初始的时候将源节点剔除掉, 而是与其他节点放在一起做 filter, 按照这个逻辑, 只要源节点的权值足够大, 是有可能成为目标节点的。

那紧接着的问题是: 如果源节点和目标节点是同一个, migrate 操作会怎样进行呢?

实验得知, nova-compute 在做 migrate 的时候会检查目标节点, 如果发现目标节点与源节点相同, 会抛出 UnableToMigrateToSelf 异常。Nova-compute 失败之后, scheduler 会重新调度, 由于有 RetryFilter, 会将之前选择的源节点过滤掉, 这样就能选到不同的计算节点了。

关于 RetryFilter, 大家还有印象吗? 如果生疏了可以看前面章节。

好, 言归正传。在上面的操作中 scheduler 选择的目標节点是 devstack-compute1, 意味着 instance 将从 devstack-controller 迁移到 devstack-compute1。

### 4. nova-scheduler 发送消息

nova-scheduler 发送消息, 通知计算节点可以迁移 instance 了。源代码在 /opt/stack/nova/nova/scheduler/filter\_scheduler.py 第 95 行, 方法为 selectdestinations, 如图 7-126 所示。

```
94
95     self.notifier.info(context,
96                        dict(request_spec=request_spec))
```

图 7-126

### 5. nova-compute 执行操作

nova-compute 会在源计算节点和目标计算节点上分别执行操作。

## (1) 源计算节点

迁移操作在源节点上首先会关闭 instance，然后将 instance 的镜像文件传到目标节点上。查看日志/opt/stack/logs/n-cpu.log，具体步骤如下：

- 开始 migrate，如图 7-127 所示。

```
2016-01-12 02:06:05.065 DEBUG nova.virt.libvirt.driver [req-d1268410-e367-419f-b5ae-f5ee00d555f9 admin admin] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddba001]
Starting migrate disk and power off migrate_disk_and_power_off /opt/stack/nova
```

图 7-127

- 在目标节点上创建 instance 的目录。

nova-compute 首先会尝试通过 ssh 在目标节点上的 instance 目录里 touch 一个临时文件，日志如图 7-128 所示。

```
2016-01-12 02:06:05.123 DEBUG nova.virt.libvirt.volume.remotefs [req-d1268410-e367-419f-b5ae-f5ee00d555f9 admin admin] Creating file /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001/6ba638d1dc814a47becc61d4edbc2280.tmp
on remote host 192.168.104.11 create_file /opt/stack/nova/nova/virt/libvirt/volume/remotefs.py:97
2016-01-12 02:06:05.123 DEBUG oslo_concurrency.processutils [req-d1268410-e367-419f-b5ae-f5ee00d555f9 admin admin] Running cmd (subprocess): ssh 192.168.104.11 touch /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001/6ba638d1dc814a47becc61d4edbc2280.tmp
execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
```

图 7-128

如果 touch 失败，说明目标节点上还没有该 instance 的目录，也就是说，源节点和目标节点没有共享存储。那么接下来就要在目标节点上创建 instance 的目录，日志如图 7-129 所示。

```
2016-01-12 02:06:25.710 DEBUG oslo_concurrency.processutils [req-d1268410-e367-419f-b5ae-f5ee00d555f9 admin admin] CMD "ssh 192.168.104.11 touch /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001/6ba638d1dc814a47becc61d4edbc2280.tmp" returned: 1 in 20.586s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
2016-01-12 02:06:25.710 DEBUG oslo_concurrency.processutils [req-d1268410-e367-419f-b5ae-f5ee00d555f9 admin admin] u ssh 192.168.104.11 touch /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001/6ba638d1dc814a47becc61d4edbc2280.tmp failed. Not Retrying. execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:328
2016-01-12 02:06:25.711 DEBUG nova.virt.libvirt.volume.remotefs [req-d1268410-e367-419f-b5ae-f5ee00d555f9 admin admin] Creating directory /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001 on remote host 192.168.104.11 create_dir /opt/stack/nova/nova/virt/libvirt/volume/remotefs.py:109
2016-01-12 02:06:25.711 DEBUG oslo_concurrency.processutils [req-d1268410-e367-419f-b5ae-f5ee00d555f9 admin admin] Running cmd (subprocess): ssh 192.168.104.11 mkdir -p /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001
execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
```

图 7-129

- 关闭 instance，如图 7-130 所示。

```
2016-01-12 02:06:46.308 DEBUG nova.virt.libvirt.driver [req-d1268410-e367-419f-b5ae-f5ee00d555f9 admin admin] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddba001] Shutting down instance from state 1 clean_shutdown /opt/stack/nova/nova/virt/libvirt/driver.py:2227
2016-01-12 02:06:49.327 INFO nova.virt.libvirt.driver [req-d1268410-e367-419f-b5ae-f5ee00d555f9 admin admin] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddba001] Instance shutdown successfully after 3 seconds.
```

图 7-130

- 将 instance 镜像文件通过 scp 传到目标节点，如 7-131 图所示。

```

419f-b5ae-f5ee00d555f9 admin admin] Running cmd (subprocess): mv /opt/stack/data/
a/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001 /opt/stack/data/nova/inst
ances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001_resize execute /usr/local/lib/python
2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-01-12 02:06:49.358 DEBUG oslo_concurrency.processutils [req-d1268410-e367-
419f-b5ae-f5ee00d555f9 admin admin] cmd "mv /opt/stack/data/nova/instances/d371
bbfb-43d8-4385-a5f7-ca3f4ddba001 /opt/stack/data/nova/instances/d371bbfb-43d8-4
385-a5f7-ca3f4ddba001_resize" returned: 0 in 0.021s execute /usr/local/lib/pyth
on2.7/dist-packages/oslo_concurrency/processutils.py:280
2016-01-12 02:06:49.359 DEBUG nova.virt.libvirt.volume.remotefs [req-d1268410-e
367-419f-b5ae-f5ee00d555f9 admin admin] Copying file /opt/stack/data/nova/inst
ances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001_resize/disk to 192.168.104.11: /opt/st
ack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001/disk copy file op
t/stack/nova/nova/virt/libvirt/volume/remotefs.py:121
2016-01-12 02:06:49.360 DEBUG oslo_concurrency.processutils [req-d1268410-e367-
419f-b5ae-f5ee00d555f9 admin admin] Running cmd (subprocess): scp /opt/stack/d
ata/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001_resize/disk 192.168.104.
11: /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001/disk exe
cute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:25
0

```

图 7-131

## (2) 目标计算节点

在目标节点上启动 instance，过程与 launch instance 非常类似。

会经过如下几个步骤：

- 为 instance 准备 CPU、内存和磁盘资源。
- 创建 instance 镜像文件。
- 创建 instance 的 XML 定义文件。
- 创建虚拟网络并启动 instance。

查看日志记录/opt/stack/logs/n-cpu.log，分析留给大家练习。

## (3) Confirm

这时，instance 会处于“Confirm or Revert Resize/Migrate”状态，需要用户确认或者退回当前的迁移操作，实际上给了用户一个反悔的机会。如图 7-132 所示。

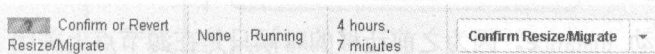


图 7-132

当我们按下 Confirm 按钮后，会发生如下事情：

nova-api 接收到 confirm 的消息，如图 7-133 所示。

```

2016-01-12 02:08:15.063 DEBUG nova.api.openstack.wsgi [req-e6d1ef6d-2433-4448-9
2d4-5f945d161856 admin admin] Action: 'action', calling method: <bound method S
erversController._action_confirm_resize of <nova.api.openstack.compute.servers.
ServersController object at 0x7f151c68ef90>.>, body: {'confirmResize': null} _pr
ocess_stack /opt/stack/nova/nova/api/openstack/wsgi.py:789

```

图 7-133

源计算节点删除 instance 的目录，并在 Hypervisor 上删除 instance。如图 7-134 所示。

```

2016-01-12 02:08:15.171 DEBUG nova.compute.manager [req-e6d1ef6d-2433-4448-92d
4-5f945d161856 admin admin] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddba001] go
ing to confirm migration 5 do_confirm_resize /opt/stack/nova/nova/compute/manag
er.py:3360
2016-01-12 02:08:15.589 DEBUG oslo_concurrency.processutils [req-e6d1ef6d-2433-
4448-92d4-5f945d161856 admin admin] cmd "rm -rf /opt/stack/data/nova/instances/
d371bbfb-43d8-4385-a5f7-ca3f4ddba001_resize" returned: 0 in 0.012s execute /us
r/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280

```

图 7-134



目标计算节点不需要做任何事情。

#### (4) Revert

如果执行的是 Revert 操作会发生什么事情呢？如图 7-135 所示。

nova-api 接收到 revert 的消息，如图 7-136 所示。

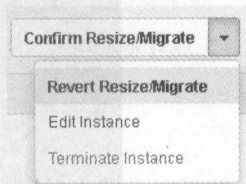


图 7-135

```
2016-01-12 18:45:01.065 DEBUG nova.api.openstack.wsgi [req-114666ba-7095-4a37-93b2-4cfca002cc82 admin admin] Action: 'action', calling method: <bound method ServersController._action_revert_resize of <nova.api.openstack.compute.servers.ServersController object at 0x7f151ce6ef90>>, body: {"revertResize": null}_process_stack /opt/stack/nova/nova/api/openstack/wsgi.py:789
```

图 7-136

在目标计算节点上关闭 instance，删除 instance 的目录，并在 Hypervisor 上删除 instance，如图 7-137 所示。

```
2016-01-12 18:45:02.303 1990 INFO nova.virt.libvirt.driver [-] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddba001] Instance destroyed successfully.

2016-01-12 18:45:02.319 DEBUG oslo_concurrency.processutils [req-114666ba-7095-4a37-93b2-4cfca002cc82 admin admin] CMD "mv /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001 /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001_del" returned: 0 in 0.014s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280

2016-01-12 18:45:02.320 INFO nova.virt.libvirt.driver [req-114666ba-7095-4a37-93b2-4cfca002cc82 admin admin] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddba001] Deleting instance files /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001_del

2016-01-12 18:45:02.321 INFO nova.virt.libvirt.driver [req-114666ba-7095-4a37-93b2-4cfca002cc82 admin admin] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddba001] Deletion of /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001_del complete
```

图 7-137

源计算节点上启动 instance。因为之前迁移的时候只是在源节点上关闭了该 instance，revert 操作只需重新启动 instance。如图 7-138 所示。

```
2016-01-12 18:45:02.972 DEBUG nova.virt.libvirt.driver [req-114666ba-7095-4a37-93b2-4cfca002cc82 admin admin] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddba001] Starting finish_revert_migration finish_revert_migration /opt/stack/nova/nova/virt/libvirt/driver.py:6884

2016-01-12 18:45:02.973 DEBUG oslo_concurrency.processutils [req-114666ba-7095-4a37-93b2-4cfca002cc82 admin admin] Running cmd (subprocess): mv /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001_resize /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001 execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250

2016-01-12 18:45:02.986 DEBUG oslo_concurrency.processutils [req-114666ba-7095-4a37-93b2-4cfca002cc82 admin admin] CMD "mv /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001_resize /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001" returned: 0 in 0.013s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280

2016-01-12 18:45:03.719 INFO nova.compute.manager [req-c787cd00-d5cd-4d85-9a26-b247dcb6f90c None None] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddba001] VM started (LifecycleEvent)

2016-01-12 18:45:03.729 2880 INFO nova.virt.libvirt.driver [-] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddba001] Instance running successfully.

2016-01-12 18:45:03.729 DEBUG nova.virt.libvirt.driver [req-114666ba-7095-4a37-93b2-4cfca002cc82 admin admin] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddba001] finish_revert_migration finished successfully. finish_revert_migration /opt/stack/nova/nova/virt/libvirt/driver.py:6919
```

图 7-138



以上是 Migrate 操作的完整流程, 这里有一点需要特别注意:

迁移过程中源和目标节点之前需要使用 ssh 和 scp, 为了使操作顺利进行, 必须要保证 nova-compute 进程的启动用户(通常是 nova, 也可能是 root, 可以通过 ps 命令确认)能够在计算节点之间无密码访问。否则 nova-compute 会等待密码输入, 但后台服务是无法输入密码的, 迁移操作会一直卡在那里。

### 7.3.16 Resize

Resize 的作用是调整 instance 的 vCPU、内存和磁盘资源。Instance 需要多少资源是定义在 flavor 中的, resize 操作是通过为 instance 选择新的 flavor 来调整资源的分配。

有了前面对 Migrate 的分析, 再来看 Resize 的实现就非常简单了。因为 instance 需要分配的资源发生了变化, 在 resize 之前需要借助 nova-scheduler 重新为 instance 选择一个合适的计算节点, 如果选择的节点与当前节点不是同一个, 那么就需要做 Migrate。

所以本质上讲: Resize 是在 Migrate 的同时应用新的 flavor。

Migrate 可以看作是 resize 的一个特例: flavor 没发生变化的 resize, 这也是为什么我们在上一节日志中看到 migrate 实际上是在执行 resize 操作。

如图 7-139 所示是 Resize instance 的流程图。

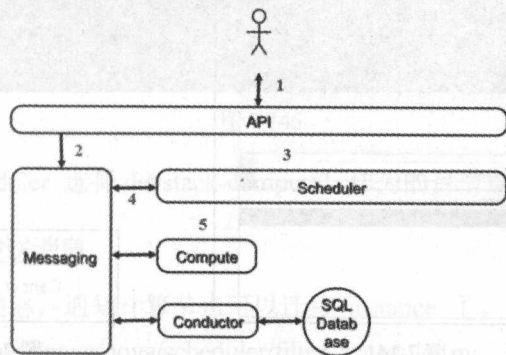


图 7-139

- 向 nova-api 发送请求。
- nova-api 发送消息。
- nova-scheduler 执行调度。
- nova-scheduler 发送消息。
- nova-compute 执行操作。

Resize 分两种情况:

nova-scheduler 选择的目标节点与源节点是不同节点。操作过程跟上一节 Migrate 几乎完全一样, 只是在目标节点启动 instance 的时候按新的 flavor 分配资源。

同时, 因为要跨节点复制文件, 也必须要保证 nova-compute 进程的启动用户(通常是 nova,

也可能是 root，可以通过 ps 命令确认）能够在计算节点之间无密码访问。

对这一种情况我们不再赘述，请参看前面 Migrate 小节。

目标节点与源节点是同一个节点。则不需要 migrate。下面我们重点讨论这一种情况。

1. 向 nova-api 发送请求

客户（可以是 OpenStack 最终用户，也可以是其他程序）向 API (nova-api) 发送请求：“帮我 Resize 这个 Instance”，如图 7-140 所示。

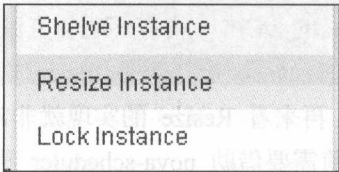


图 7-140

选择新的 flavor，如图 7-141 所示。

单击“Resize”，如图 7-142 所示。

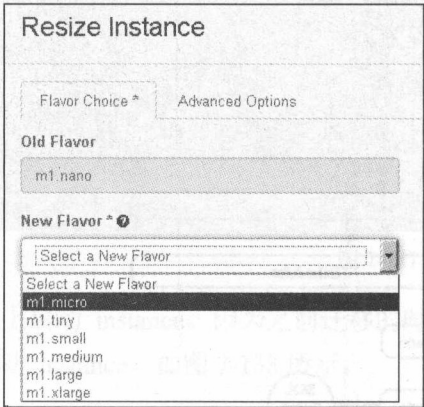


图 7-141

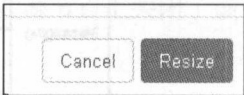


图 7-142

查看日志 /opt/stack/logs/n-api.log，如图 7-143 所示。

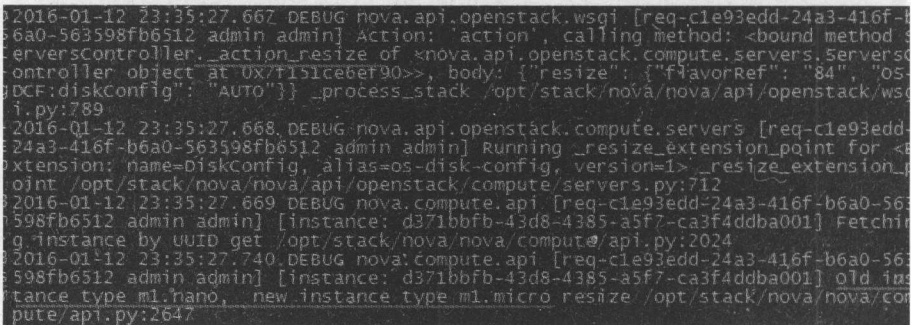


图 7-143

## 2. nova-api 发送消息

nova-api 向 Messaging (RabbitMQ) 发送了一条消息：“Resize 这个 Instance”

查看源代码 /opt/stack/nova/nova/compute/api.py, 方法是 `resize_instance`, 如图 7-144 所示。

```
2709 scheduler_hint = {
2710     self.compute_task_api.resize_instance(context, instance,
2711     extra_instance_updates, scheduler_hint=scheduler_hint,
2712     flavor=new_instance_type,
2713     reservations=quotas.reservations or [],
2714     clean_shutdown=clean_shutdown)
2715 }
```

图 7-144

## 3. nova-scheduler 执行调度

nova-scheduler 收到消息后, 会为 instance 选择合适的目标计算节点。

查看日志 /opt/stack/logs/n-sch.log, 如图 7-145 所示。

```
#2016-01-12 23:35:27.875 DEBUG nova.scheduler.filter_scheduler [req-c1e93edd-24
3-416f-b6a0-563598fb6512 admin admin] Filtered [(devstack-compute1, devstack-co
mpute1) ram:9424 disk:4096 io_ops:0 instances:1, (devstack-controller, devstack-
controller) ram:9296 disk:4096 io_ops:0 instances:3] _schedule /opt/stack/nova
/nova/scheduler/filter_scheduler.py:152
#2016-01-12 23:35:27.875 DEBUG nova.scheduler.filter_scheduler [req-c1e93edd-24
3-416f-b6a0-563598fb6512 admin admin] weighed [weighedHost [host: (devstack-co
mpute1, devstack-compute1) ram:9424 disk:4096 io_ops:0 instances:1, weight: 0.9
3254637437], weighedHost [host: (devstack-controller, devstack-controller) ram
9296 disk:4096 io_ops:0 instances:3, weight: 0.97976391231]] _schedule /opt/st
ack/nova/nova/scheduler/filter_scheduler.py:157
#2016-01-12 23:35:27.875 DEBUG nova.scheduler.filter_scheduler [req-c1e93edd-24
3-416f-b6a0-563598fb6512 admin admin] selected host: weighedHost [host: (devst
ack-compute1, devstack-compute1) ram:9424 disk:4096 io_ops:0 instances:1, weigh
t: 0.993254637437] _schedule /opt/stack/nova/nova/scheduler/filter_scheduler.py
167
```

图 7-145

在本例中, nova-scheduler 选择 devstack-compute1 作为目标节点, 与源节点相同。

## 4. nova-scheduler 发送消息

nova-scheduler 发送消息, 通知计算节点可以迁移 instance 了。

源代码在 /opt/stack/nova/nova/scheduler/filter\_scheduler.py 第 95 行, 方法为 `select_destinations`, 如图 7-146 所示。

```
94
95 self.notifier.info(context,
96 dict(request_spec=request_spec))
```

图 7-146

## 5. nova-compute 执行操作

在目标节点上启动 instance, 过程与 `launch instance` 非常类似。其日志记录在 /opt/stack/logs/n-cpu.log 中。

会经过如下几个步骤:

按新的 flavor 为 instance 准备 CPU、内存和磁盘资源, 如图 7-147 所示。

```

2016-01-12 23:35:28.449 DEBUG oslo_concurrency.lockutils [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] Lock "compute_resources" acquired by "nova.compute_resource_tracker.resize_claim" : waited 0.000s inner /usr/local/lib/python2.7/dist-packages/oslo_concurrency/lockutils.py:253
2016-01-12 23:35:28.467 DEBUG nova.compute_resource_tracker [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] Memory overhead for 128 MB instance; 0 MB to move_claim /opt/stack/nova/nova.compute/resource_tracker.py:244
2016-01-12 23:35:28.472 INFO nova.compute.claims [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddb001] Attempting claim: memory 128 MB, disk 0 GB
2016-01-12 23:35:28.472 INFO nova.compute.claims [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddb001] Total memory: 10000 MB, used: 576.00 MB
2016-01-12 23:35:28.473 INFO nova.compute.claims [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddb001] memory limit: 15000.00 MB, free: 14424.00 MB
2016-01-12 23:35:28.473 INFO nova.compute.claims [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddb001] Total disk: 9 GB, used: 0.00 GB
2016-01-12 23:35:28.473 INFO nova.compute.claims [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddb001] disk limit: 9.00 GB, free: 9.00 GB
2016-01-12 23:35:28.487 DEBUG nova.compute.resources.vcpu [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] Total CPUs: 6 vCPUs, used: 1.00 vCPUs test /opt/stack/nova/nova.compute/resources/vcpu.py:52
2016-01-12 23:35:28.487 DEBUG nova.compute.resources.vcpu [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] CPUs limit not specified, defaulting to unlimited test /opt/stack/nova/nova.compute/resources/vcpu.py:56
2016-01-12 23:35:28.487 INFO nova.compute.claims [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddb001] Claim successful

```

图 7-147

关闭 instance，如图 7-148 所示。

```

2016-01-12 23:35:29.019 DEBUG nova.virt.libvirt.driver [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddb001] Shutting down instance from state 1 clean_shutdown /opt/stack/nova/nova.virt/libvirt/driver.py:2227
2016-01-12 23:35:32.046 INFO nova.virt.libvirt.driver [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddb001] Instance shutdown successfully after 3 seconds.

```

图 7-148

创建 instance 镜像文件，如图 7-149 所示。

```

2016-01-12 23:35:33.015 INFO nova.virt.libvirt.driver [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddb001] Creating image

```

图 7-149

将 instance 的目录备份一份，命名为 <instance\_id>\_resize，以便 revert，如图 7-150 所示。

```

416f-b6a0-563598fb6512 admin admin] Running cmd (subprocess): mv /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddb001 /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddb001_resize execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-01-12 23:35:32.079 DEBUG oslo_concurrency.processutils [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] CMD "mv /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddb001 /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddb001_resize" returned: 0 in 0.018s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
2016-01-12 23:35:32.080 DEBUG oslo_concurrency.processutils [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] Running cmd (subprocess): mkdir -p /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddb001 execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-01-12 23:35:32.093 DEBUG oslo_concurrency.processutils [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] CMD "mkdir -p /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddb001" returned: 0 in 0.013s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
2016-01-12 23:35:32.094 DEBUG oslo_concurrency.processutils [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] Running cmd (subprocess): cp /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddb001 /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddb001_resize disk execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-01-12 23:35:32.120 DEBUG oslo_concurrency.processutils [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] CMD "cp /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddb001 /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddb001_resize" returned: 0 in 0.026s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280

```

图 7-150



创建 instance 的 XML 定义文件, 如图 7-151 所示。

```
2016-01-12 23:35:33.222 DEBUG nova.virt.libvirt.config [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] Generated XML (<?xml version="1.0" encoding="UTF-8" domain type="qemu">\n <uuid>d371bbfb-43d8-4385-a5f7-ca3f4ddba001</uuid>\n <name>instance-00000008</name>\n <memory>131072</memory>\n <vcpu>1</vcpu>\n <metadata>\n <nova:instance xml:ns:nova="http://openstack.org/xmlns/libvirt/nova/1.0">\n <nova:package ver
```

图 7-151

准备虚拟网络, 如图 7-152 所示。

```
2016-01-12 23:35:33.225 DEBUG nova.virt.libvirt.vif [req-c1e93edd-24a3-416f-b6a0-563598fb6512 admin admin] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddba001] Ensuring bridge brq5333bf1-67 plug_bridge /opt/stack/nova/nova/virt/libvirt/vif.py:482
```

图 7-152

启动 instance, 如图 7-153 所示。

```
2016-01-12 23:35:33.908 INFO nova.compute.manager [req-5a05af43-fc75-4f19-99e6-b9eb7e1a6e6a None None] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddba001] VM Started (Lifecycle Event)
2016-01-12 23:35:33.912 1990 INFO nova.virt.libvirt.driver [-] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddba001] Instance running successfully.
```

图 7-153

### (1) Confirm

这时, instance 的状态处于“Confirm or Revert Resize/Migrate”状态, 需要用户确认或者退回当前的迁移操作, 实际上给了用户一个反悔的机会, 如图 7-154 所示。

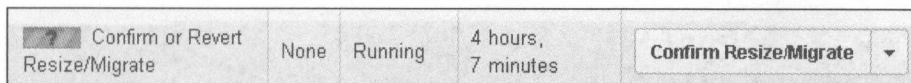


图 7-154

当我们按下 Confirm 按钮后, 会发生如下事情:

nova-api 接收到 confirm 的消息, 如图 7-155 所示。

```
2016-01-13 00:02:32.922 DEBUG nova.api.openstack.wsgi [req-104b8b50-6ae8-47c9-8c76-bd7c1a038729 admin admin] Action: 'action', calling method: <bund method 5, serverscontroller._action_confirm_resize of <nova.api.openstack.compute.servers.ServersController object at 0x7f151ce6ef90>.> body: {"confirmResize": null} _process_stack /opt/stack/nova/nova/api/openstack/wsgi.py:789
```

图 7-155

删除计算节点上备份的 instance 目录 <instance\_id>\_resize, 如图 7-156 所示。

```
2016-01-13 00:02:33.329 DEBUG nova.compute.manager [req-104b8b50-6ae8-47c9-8c76-bd7c1a038729 admin admin] [instance: d371bbfb-43d8-4385-a5f7-ca3f4ddba001] Going to confirm migration 14 do_confirm_resize /opt/stack/nova/nova/compute/manager.py:3360
2016-01-13 00:02:33.816 DEBUG oslo_concurrency.processutils [req-104b8b50-6ae8-47c9-8c76-bd7c1a038729 admin admin] Running cmd (subprocess): rm -rf /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001_resize execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-01-13 00:02:33.830 DEBUG oslo_concurrency.processutils [req-104b8b50-6ae8-47c9-8c76-bd7c1a038729 admin admin] CMD "rm -rf /opt/stack/data/nova/instances/d371bbfb-43d8-4385-a5f7-ca3f4ddba001_resize" returned: 0 in 0.014s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
```

图 7-156

## (2) Revert

反过来, 如果执行 Revert 操作会发生什么事情呢? 如 7-157 图所示。

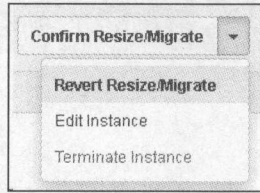


图 7-157

nova-api 接收到 revert 的消息, 如图 7-158 所示。

```
2016-01-13 00:24:03.458 DEBUG nova.api.openstack.wsgi [req-5f2b0d4b-c131-48f3-95eb-ce721abae839 admin admin] Action: 'action', calling method; <bound method ServersController._action_revert_resize of <nova.api.openstack.compute.servers.ServersController object at 0x7f151ce6ef90>>; body: {'revertResize': null} _process_stack /opt/stack/nova/nova/api/openstack/wsgi.py:789
```

7-158

在计算节点上关闭 instance, 如图 7-159 所示。

```
2016-01-13 00:24:04.748 INFO nova.virt.libvirt.driver [-] [instance: d371bbfb-bfb-43d8-4385-a5f7-ca3f4ddba001] Instance destroyed successfully.
```

图 7-159

通过备份目录 <instance\_id>\_resize 恢复 instance 目录, 如图 7-160 所示。

```
2016-01-13 00:24:05.341 DEBUG nova.virt.libvirt.driver [req-5f2b0d4b-c131-48f3-95eb-ce721abae839 admin admin] [instance: d371bbfb-bfb-43d8-4385-a5f7-ca3f4ddba001] Starting finish_revert_migration/finish_revert_migration /opt/stack/nova/nova/virt/libvirt/driver.py:6884
2016-01-13 00:24:05.342 DEBUG oslo_concurrency.processutils [req-5f2b0d4b-c131-48f3-95eb-ce721abae839 admin admin] Running cmd (subprocess): mv /opt/stack/data/nova/instances/d371bbfb-bfb-43d8-4385-a5f7-ca3f4ddba001_resize /opt/stack/data/nova/instances/d371bbfb-bfb-43d8-4385-a5f7-ca3f4ddba001 'execute' /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-01-13 00:24:05.354 DEBUG oslo_concurrency.processutils [req-5f2b0d4b-c131-48f3-95eb-ce721abae839 admin admin] CMD 'mv /opt/stack/data/nova/instances/d371bbfb-bfb-43d8-4385-a5f7-ca3f4ddba001_resize /opt/stack/data/nova/instances/d371bbfb-bfb-43d8-4385-a5f7-ca3f4ddba001' returned: 0 in 0.012s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
```

图 7-160

重新启动 instance, 如图 7-161 所示。

```
2016-01-13 00:24:06.078 INFO nova.compute.manager [req-5a05af43-fc75-4f19-99e6-b9eb7e1a6e6a None None] [instance: d371bbfb-bfb-43d8-4385-a5f7-ca3f4ddba001] VM started (lifecycle Event)
2016-01-13 00:24:06.083 INFO nova.virt.libvirt.driver [-] [instance: d371bbfb-bfb-43d8-4385-a5f7-ca3f4ddba001] Instance running successfully.
2016-01-13 00:24:06.083 DEBUG nova.virt.libvirt.driver [req-5f2b0d4b-c131-48f3-95eb-ce721abae839 admin admin] [instance: d371bbfb-bfb-43d8-4385-a5f7-ca3f4ddba001] finish_revert_migration finished successfully. finish_revert_migration /opt/stack/nova/nova/virt/libvirt/driver.py:6919
```

图 7-161

### 7.3.17 Live Migrate

migrate 操作会先将 instance 停掉, 也就是所谓的“冷迁移”。而 Live Migrate 是“热迁移”,

也叫“在线迁移”，instance 不会停机。

Live Migrate 分两种：

- 源和目标节点没有共享存储，instance 在迁移的时候需要将其镜像文件从源节点传到目标节点，这叫做 Block Migration（块迁移）
- 源和目标节点共享存储，instance 的镜像文件不需要迁移，只需要将 instance 的状态迁移到目标节点。

源和目标节点需要满足一些条件才能支持 Live Migration：

- (1) 源和目标节点的 CPU 类型要一致。
- (2) 源和目标节点的 Libvirt 版本要一致。
- (3) 源和目标节点能相互识别对方的主机名称，比如可以在 /etc/hosts 中加入对方的条目，如图 7-162 所示。

```
192.168.104.10 devstack-controller
192.168.104.11 devstack-compute1
```

图 7-162

- (4) 在源和目标节点的 /etc/nova/nova.conf 中指明在线迁移时使用 TCP 协议，如图 7-163 所示。

```
[libvirt]
inject_partition = -2
live_migration_uri = qemu+tcp://stack@s/system
use_usb_tablet = False
cpu_mode = none
virt_type = qemu
```

图 7-163

- (5) Instance 使用 config driver 保存其 metadata。在 Block Migration 过程中，该 config driver 也需要迁移到目标节点。由于目前 libvirt 只支持迁移 vfat 类型的 config driver，所以必须在 /etc/nova/nova.conf 中明确指明 launch instance 时创建 vfat 类型的 config driver，如图 7-164 所示。

```
[DEFAULT]
config_drive_format = vfat
vif_plugging_timeout = 300
vif_plugging_is_fatal = True
```

图 7-164

- (6) 源和目标节点的 Libvirt TCP 远程监听服务得打开，需要在下面两个配置文件中做一点配置。

- /etc/default/libvirt-bin

```
start_libvirtd="yes"
```

```
libvirtd_opts="-d -l"
```

- /etc/libvirt/libvirtd.conf

```
listen_tls = 0
listen_tcp = 1
unix_sock_group = "libvirtd"
unix_sock_ro_perms = "0777"
unix_sock_rw_perms = "0770"
auth_unix_ro = "none"
auth_unix_rw = "none"
auth_tcp = "none"
```

(7) 然后重启 Libvirtd 服务

```
service libvirt-bin restart
```

## 1. 非共享存储 Block Migration

我们先讨论非共享存储的 Block Migration，如图 7-165 所示流程图。

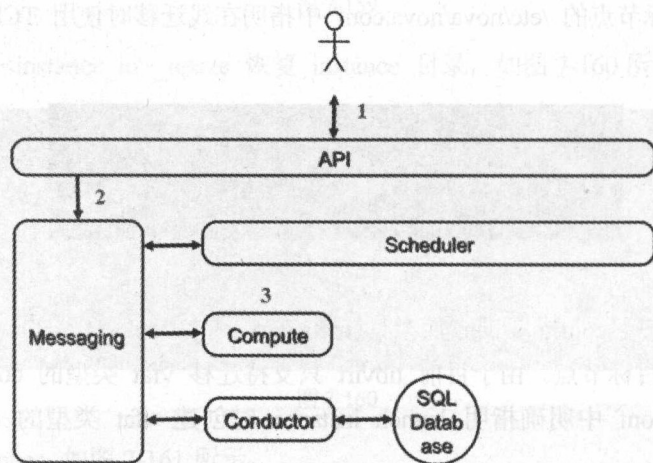


图 7-165

- 向 nova-api 发送请求。
- nova-api 发送消息。
- nova-compute 执行操作。

下面我们详细讨论每一个步骤。

### (1) 向 nova-api 发送请求

客户（可以是 OpenStack 最终用户，也可以是其他程序）向 API（nova-api）发送请求：“帮我将这个 Instance 从节点 A Live Migrate 到节点 B”，如图 7-166 所示。



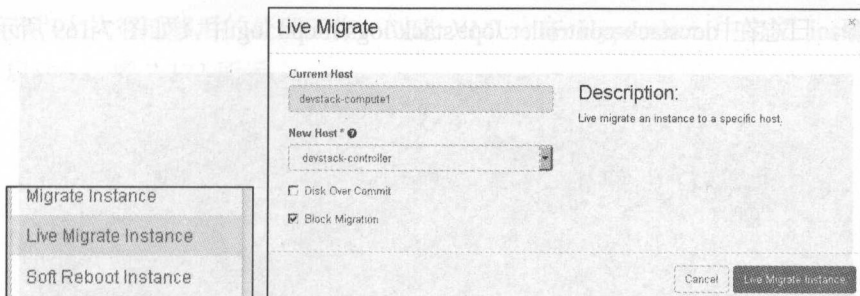


图 7-166

这里源节点是 `devstack-compute1`，目标节点是 `devstack-controller`，因为是非共享存储，记得将“Block Migration”选中上。

这里还有一个“Disk Over Commit”选项，如果选中了此选项，nova 在检查目标节点的磁盘空间是否足够时，是以 instance 磁盘镜像文件定义的最大容量为准；否则，以磁盘镜像文件当前的实际大小为准。

查看日志 `/opt/stack/logs/n-api.log`，如图 7-167 所示。

```
2016-01-17 22:44:00.597 DEBUG nova.api.openstack.wsgi [req-a77a8836-493c-48d6-8449-f56af6762968 admin admin] Action: 'action', calling method: <bound method Migrateservercontroller._migrate_live of <nova.api.openstack.compute.migrate_server.Migrateservercontroller object at 0x7f151cd36b50>>. body: {'os-migrateLive': {'disk_over_commit': false, 'block_migration': true, 'host': 'devstack-controller'}} _process_stack /opt/stack/nova/nova/api/openstack/wsgi.py:789
2016-01-17 22:44:00.600 DEBUG nova.compute.api [req-a77a8836-493c-48d6-8449-f56af6762968 admin admin] [instance: 84f81b69-8937-4d8e-b16d-0f67e9b389a3] Fetching instance by UUID get /opt/stack/nova/nova/compute/api.py:2024
2016-01-17 22:44:00.665 DEBUG nova.compute.api [req-a77a8836-493c-48d6-8449-f56af6762968 admin admin] [instance: 84f81b69-8937-4d8e-b16d-0f67e9b389a3] Going to try to live migrate instance to devstack-controller live_migrate /opt/stack/nova/nova/compute/api.py:3289
```

图 7-167

## (2) nova-api 发送消息

nova-api 向 Messaging (RabbitMQ) 发送了一条消息：“Live Migrate 这个 Instance”

查看源代码 `/opt/stack/nova/nova/compute/api.py`，方法是 `live_migrate`，如图 7-168 所示。

```
2283 def live_migrate(self, context, instance, block_migration,
2284                 disk_over_commit, host_name):
2285     LOG.debug(
2286         '%s: Live migrate instance %s to %s', instance=instance)
2287     instance.task_state = task_states.MIGRATING
2288     instance.save(expected_task_state=[None])
2289     self._record_action_start(context, instance,
2290                             instance_actions.LIVE_MIGRATION)
2291     self.compute_task_api.live_migrate_instance(context, instance,
2292         host_name, block_migration=block_migration,
2293         disk_over_commit=disk_over_commit)
2294
```

图 7-168

## (3) nova-compute 执行操作

源和目标节点执行 Live Migrate 的操作过程如下：

目标节点执行迁移前的准备工作，首先将 instance 的数据迁移过来，主要包括镜像文件、虚

拟网络等资源，日志在 `devstack-controller:/opt/stack/logs/n-cpu.log` 中，如图 7-169 所示。

```
2016-01-17 22:44:01.572 DEBUG nova.virt.libvirt.driver [req-a77a8836-493c-4
8d6-8449-f56af6762968 admin admin] [instance: 84f81b69-8937-4d8e-b16d-0f67e
9b389a3] migrate_data in pre_live_migration: {u'disk_over_commit': False, u'
disk_available_mb': 3072, u'image_type': u'default', u'is_shared_instance_
path': False, u'filename': u'tmpr31064', u'instance_relative_path': u'84f81
b69-8937-4d8e-b16d-0f67e9b389a3', u'block_migration': True, u'is_shared_blo
ck_storage': False, u'is_volume_backed': False, u'is_shared_storage': False
} pre_live_migration /opt/stack/nova/nova/virt/libvirt.driver.py:6157
2016-01-17 22:44:01.573 DEBUG nova.virt.libvirt.driver [req-a77a8836-493c-4
8d6-8449-f56af6762968 admin admin] [instance: 84f81b69-8937-4d8e-b16d-0f67e
9b389a3] creating instance directory: /opt/stack/data/nova/instances/84f81b
69-8937-4d8e-b16d-0f67e9b389a3 pre_live_migration /opt/stack/nova/nova/virt
/libvirt.driver.py:6190
2016-01-17 22:44:01.574 DEBUG nova.virt.libvirt.driver [req-a77a8836-493c-4
8d6-8449-f56af6762968 admin admin] [instance: 84f81b69-8937-4d8e-b16d-0f67e
9b389a3] checking to make sure images and backing files are present before
live_migration. pre_live_migration /opt/stack/nova/nova/virt/libvirt.driver
.py:6196
2016-01-17 22:44:01.698 DEBUG nova.virt.libvirt.driver [req-a77a8836-493c-4
8d6-8449-f56af6762968 admin admin] [instance: 84f81b69-8937-4d8e-b16d-0f67e
9b389a3] Plugging vifs before live migration. pre_live_migration /opt/stack
/nova/nova/virt/libvirt.driver.py:6234
```

图 7-169

源节点启动迁移操作，暂停 instance，如图 7-170 所示。

```
2016-01-17 22:44:02.038 DEBUG nova.virt.libvirt.driver [req-a77a8836-493c-4
8d6-8449-f56af6762968 admin admin] [instance: 84f81b69-8937-4d8e-b16d-0f67e
9b389a3] Starting monitoring of live migration _live_migration /opt/stack/n
ova/nova/virt/libvirt.driver.py:6067
2016-01-17 22:44:02.038 DEBUG nova.objects.instance [req-a77a8836-493c-48d6
-8449-f56af6762968 admin admin] Lazy-loading flavor on instance uuid 84f8
1b69-8937-4d8e-b16d-0f67e9b389a3 obj_load_attr /opt/stack/nova/nova/objects
/instance.py:860
2016-01-17 22:44:02.109 INFO nova.virt.libvirt.driver [req-a77a8836-493c-48
d6-8449-f56af6762968 admin admin] [instance: 84f81b69-8937-4d8e-b16d-0f67e9
b389a3] Increasing downtime to 46ms after 0 sec elapsed time
2016-01-17 22:44:02.211 INFO nova.virt.libvirt.driver [req-a77a8836-493c-48
d6-8449-f56af6762968 admin admin] [instance: 84f81b69-8937-4d8e-b16d-0f67e9
b389a3] Migration running for 0 secs, memory 100% remaining; (bytes process
ed=0, remaining=0, total=0)
2016-01-17 22:44:03.339 DEBUG nova.virt.driver [req-a6b7edce-cdb6-4ae6-adcc
-8572fc1b6512 None None] Emitting event <LifecycleEvent: 1453041843.34; 84f
81b69-8937-4d8e-b16d-0f67e9b389a3 => Paused>, emit_event /opt/stack/nova/nov
a/virt.driver.py:1303
2016-01-17 22:44:03.340 INFO nova.compute.manager [req-a6b7edce-cdb6-4ae6-a
dcc-8572fc1b6512 None None] [instance: 84f81b69-8937-4d8e-b16d-0f67e9b389a3
] VM Paused (Lifecycle Event)
```

图 7-170

在目标节点上 Resume instance，如图 7-171 所示。

```
2016-01-17 22:44:03.107 INFO nova.compute.manager [req-e5d792a9-a8af-4c86-8
0c9-25f76d466adb None None] [instance: 84f81b69-8937-4d8e-b16d-0f67e9b389a3
] VM Resumed (Lifecycle Event)
```

图 7-171

在源节点上执行迁移后的处理工作，删除 instance，如图 7-172 所示。

```
2016-01-17 22:44:03.889 INFO nova.virt.libvirt.driver [req-a77a8836-493c-4
8d6-8449-f56af6762968 admin admin] [instance: 84f81b69-8937-4d8e-b16d-0f67e
9b389a3] Migration operation has completed
2016-01-17 22:44:03.890 INFO nova.compute.manager [req-a77a8836-493c-48d6-
449-f56af6762968 admin admin] [instance: 84f81b69-8937-4d8e-b16d-0f67e9b38
9a3] _post_live_migration() is started..
2016-01-17 22:44:04.212 INFO nova.virt.libvirt.driver [req-a77a8836-493c-48
d6-8449-f56af6762968 admin admin] [instance: 84f81b69-8937-4d8e-b16d-0f67e9
b389a3] Deleting instance files /opt/stack/data/nova/instances/84f81b69-893
7-4d8e-b16d-0f67e9b389a3_del
2016-01-17 22:44:04.212 INFO nova.virt.libvirt.driver [req-a77a8836-493c-48
d6-8449-f56af6762968 admin admin] [instance: 84f81b69-8937-4d8e-b16d-0f67e9
b389a3] Deletion of /opt/stack/data/nova/instances/84f81b69-8937-4d8e-b16d-
0f67e9b389a3_del complete
```

图 7-172

在目标节点上执行迁移后的处理工作，创建 XML，在 Hypervisor 中定义 instance，使之下次能够正常启动，如图 7-173 所示。

```
2016-01-17 22:44:04.766 DEBUG nova.virt.libvirt.config [req-a77a8836-493c-48d6-8449-f56af6762968 admin admin] generated XML ('<domain type="qemu">\n
<uuid>84f81b69-8937-4d8e-b16d-0f67e9b389a3</uuid>\n <name>instance-00000000
f</name>\n <memory>65536</memory>\n <vcpu>1</vcpu>\n <metadata>\n <ric
va:instance xmlns:nova="http://openstack.org/xmlns/libvirt/nova/1.0">\n
```

图 7-173

Instance 在 Live Migrate 的整个过程中不会停机，我们通过 Ping 操作来观察，如图 7-174 所示。

```
64 bytes from 10.10.1.3: seq=15 ttl=64 time=1.019 ms
64 bytes from 10.10.1.3: seq=16 ttl=64 time=1.739 ms
64 bytes from 10.10.1.3: seq=17 ttl=64 time=1.641 ms
64 bytes from 10.10.1.3: seq=18 ttl=64 time=1.458 ms
64 bytes from 10.10.1.3: seq=19 ttl=64 time=2.256 ms
64 bytes from 10.10.1.3: seq=20 ttl=64 time=2.749 ms
64 bytes from 10.10.1.3: seq=21 ttl=64 time=9.682 ms
64 bytes from 10.10.1.3: seq=22 ttl=64 time=2.781 ms
64 bytes from 10.10.1.3: seq=23 ttl=64 time=0.523 ms
64 bytes from 10.10.1.3: seq=24 ttl=64 time=0.522 ms
64 bytes from 10.10.1.3: seq=25 ttl=64 time=0.886 ms
64 bytes from 10.10.1.3: seq=26 ttl=64 time=2.163 ms
64 bytes from 10.10.1.3: seq=27 ttl=64 time=0.580 ms
```

图 7-174

可见在迁移过程中，Ping 进程没有中断，只是有一个 ping 包的延迟增加了。下面我们再来看源和目标节点共享存储下的 Live Migrate。

## 2. 共享存储 Live Migration

有多种方式可以实现共享存储，比如可以将 instance 的镜像文件放在 NFS 服务器上，或者使用 NAS 服务器，或者分布式文件系统。

作为学习和实验，这里我们采用 NFS 方案。其他共享存储方案对于 Live Migration 本质上是一样的，只是在性能和高可用性上更好。

搭建 NFS 环境

将 devstack-controller 作为 NFS 服务器，共享其目录 /opt/stack/data/nova/instances。

devstack-compute1 作为 NFS 客户端将此目录 mount 到本机，如图 7-175 所示。

```
root@devstack-compute1:~# mount|grep nfs
devstack-controller:/opt/stack/data/nova/instances on /opt/stack/data/nova/i
nstances type nfs (rw,vers=4,addr=192.168.104.10,clientaddr=192.168.104.11)
root@devstack-compute1:~#
```

图 7-175

这样，OpenStack 的 instance 在 devstack-controller 和 devstack-compute1 上就实现共享存储了。

共享存储的迁移过程与 Block Migrate 基本上一样，只是几个环节有点区别：

向 nova-api 提交请求的时候，不能选中“Block Migrate”，如图 7-176 所示。

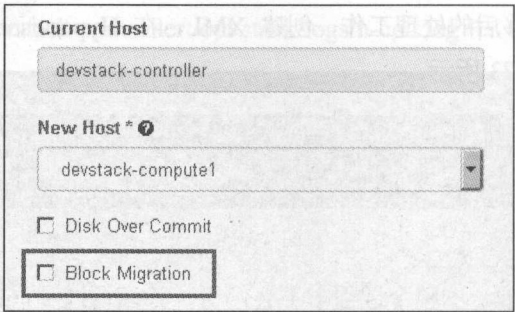


图 7-176

因为源和目标节点都能直接访问 instance 的镜像，所以目标节点在准备阶段不需要传输镜像文件，源节点在迁移后处理阶段也无须删除 instance 的目录。

只有 instance 的状态需要从源节点传输到的目标节点，整个迁移速度比 Block Migration 快很多。

具体的日志分析留个大家练习。

7.3.18 Evacuate

Rebuild 可以恢复损坏的 instance。

那如果是宿主机坏了怎么办呢？比如硬件故障或者断电造成整台计算节点无法工作，该节点上运行的 instance 如何恢复呢？用 Shelve 或者 Migrate 可不可以？很不幸，这两个操作都要求 instance 所在计算节点的 nova-compute 服务正常运行。

幸运的是，还有 Evacuate 操作。Evacuate 可在 nova-compute 无法工作的情况下将节点上的 instance 迁移到其他计算节点上。但有个前提：Instance 的镜像文件必须放在共享存储上。

如图 7-177 所示是 Evacuate instance 的流程图。

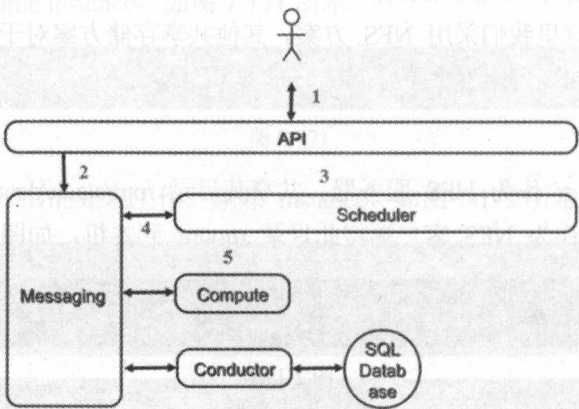


图 7-177

- 向 nova-api 发送请求。
- nova-api 发送消息。



- nova-scheduler 执行调度。
- nova-scheduler 发送消息。
- nova-compute 执行操作。

下面我们详细讨论每一个步骤。

1. 向 nova-api 发送请求

我们的实验场景如下：

Instance c2 运行在 devstack-compute1 上，如图 7-178 所示。

<input type="checkbox"/>	Project	Host	Name	Image Name	IP Address
<input type="checkbox"/>	admin	devstack-compute1	c2	cirros	10.10.1.20

图 7-178

通过断电模拟计算节点故障，然后执行 Evacuate 操作恢复 instance c2。

目前 Evacuate 只能通过 CLI 执行，如图 7-179 所示。

```
root@devstack-controller:~# nova help evacuate
usage: nova evacuate [--password <password>] [--on-shared-storage]
       <server> [<host>]

Evacuate server from failed host.

Positional arguments:
  <server>              Name or ID of server.
  <host>                Name or ID of the target host. If no host is
                        specified, the scheduler will choose one.

Optional arguments:
  --password <password> Set the provided admin password on the evacuated
                        server. Not applicable with on-shared-storage flag.
  --on-shared-storage    Specifies whether server files are located on shared
                        storage.

root@devstack-controller:~# nova evacuate c2 --on-shared-storage
+-----+-----+
| Property | value |
+-----+-----+
| adminPass | -     |
+-----+-----+
root@devstack-controller:~#
```

图 7-179

这里需要指定 --on-shared-storage 这个参数。

查看日志 /opt/stack/logs/n-api.log，如图 7-180 所示。

```
2016-01-20 17:06:28.408 DEBUG nova.api.openstack.wsgi [req-109ab176-e798-4046-9a79-6983845772bd admin admin] Action: 'action', calling
method: <bound method EvacuateController._evacuate of <nova.api.op
enstack.compute.evacuate.EvacuateController object at 0x7f151cd40f1
0>>; body: {"evacuate": {"onsharedStorage": true}} _process_stack /
opt/stack/nova/nova/api/openstack/wsgi.py:789
2016-01-20 17:06:28.409 DEBUG nova.compute.api [req-109ab176-e798-4
046-9a79-6983845772bd admin admin] [instance: 3aebf982-09f0-4b6a-b7
63-9ce594277963] Fetching instance by UUID get /opt/stack/nova/nova
/compute/api.py:2024
2016-01-20 17:06:28.458 DEBUG nova.compute.api [req-109ab176-e798-4
046-9a79-6983845772bd admin admin] [instance: 3aebf982-09f0-4b6a-b7
63-9ce594277963] vm evacuation scheduled evacuate /opt/stack/nova/n
ova/compute/api.py:3316
```

图 7-180

## 2. nova-api 发送消息

nova-api 向 Messaging (RabbitMQ) 发送了一条消息：“Evacuate 这个 Instance”  
查看源代码 /opt/stack/nova/nova/compute/api.py, 方法是 evacuate。如图 7-181 所示。

```

3343         return self.compute_task_api.rebuild_instance(context,
3344                                                         instance=instance,
3345                                                         new_pass=admin_password,
3346                                                         injected_files=None,
3347                                                         image_ref=None,
3348                                                         orig_image_ref=None,
3349                                                         orig_sys_metadata=None,
3350                                                         bdms=None,
3351                                                         recreate=True,
3352                                                         on_shared_storage=on_shared_storage,
3353                                                         host=host)

```

图 7-181

大家注意到没有, evacuate 实际上是通过 rebuild 操作实现的。  
这是可以理解的, 因为 evacuate 是用共享存储上 instance 的镜像文件重新创建虚拟机。

## 3. nova-scheduler 执行调度

nova-scheduler 收到消息后, 会为 instance 选择合适的计算节点。

查看日志 /opt/stack/logs/n-sch.log, 如图 7-182 所示。

```

2016-01-20 17:06:28.629 DEBUG nova.filters [req-109ab176-e798-4046-9a79-6983845772bd admin:admin] Filter serverGroupAffinityFilter returned 1 host(s) get_filtered_objects /opt/stack/nova/nova/filters.py:104
2016-01-20 17:06:28.629 DEBUG nova.scheduler.filter_scheduler [req-109ab176-e798-4046-9a79-6983845772bd admin:admin] Filtered [(devstack-controller, devstack-controller) ram:9232 disk:3072 io_ops:1 instances:4] _schedule /opt/stack/nova/nova/scheduler/filter_scheduler.py:152
2016-01-20 17:06:28.630 DEBUG nova.scheduler.filter_scheduler [req-109ab176-e798-4046-9a79-6983845772bd admin:admin] weighed [weighedHost [host: (devstack-controller, devstack-controller) ram:9232 disk:3072 io_ops:1 instances:4, weight: 0.0]] _schedule /opt/stack/nova/nova/scheduler/filter_scheduler.py:157
2016-01-20 17:06:28.630 DEBUG nova.scheduler.filter_scheduler [req-109ab176-e798-4046-9a79-6983845772bd admin:admin] Selected host: weighedHost [host: (devstack-controller, devstack-controller) ram:9232 disk:3072 io_ops:1 instances:4, weight: 0.0] _schedule /opt/stack/nova/nova/scheduler/filter_scheduler.py:167

```

图 7-182

nova-scheduler 最后选择在 devstack-controller 计算节点上重建 instance。

## 4. nova-scheduler 发送消息

nova-scheduler 发送消息, 通知计算节点可以创建 instance 了。

源代码在 /opt/stack/nova/nova/scheduler/filter\_scheduler.py 的第 95 行, 方法为 select\_destinations, 如图 7-183 所示。

```

94         self.notifier.info(context,
95                             "Instance can be created on %s" % (request_spec=request_spec))
96

```

图 7-183

## 5. nova-compute 执行操作

计算节点上的工作是用共享存储上的镜像文件重建 instance。日志在 `devstack-controller:/opt/stack/logs/n-cpu.log` 中。

(1) 为 instance 分配资源，如图 7-184 所示。

```
2016-01-20 17:06:28.669 INFO nova.compute.manager [req-109ab176-e798-4046-9a79-6983845772bd admin admin] [instance: 3aebf982-09f0-4b6a-b763-9ce594277963] Rebuilding instance
2016-01-20 17:06:28.670 DEBUG oslo.concurrency.lockutils [req-109ab176-e798-4046-9a79-6983845772bd admin admin] Lock "compute_resource" acquired by "nova.compute.resource_tracker.rebuild_claim" : waited 0.000s inner /usr/local/lib/python2.7/dist-packages/oslo_concurrency/lockutils.py:233
2016-01-20 17:06:28.688 DEBUG nova.compute.resource_tracker [req-109ab176-e798-4046-9a79-6983845772bd admin admin] Memory overhead for 64 MB instance: 0 MB _move_claim /opt/stack/nova/nova/compute/resource_tracker.py:244
2016-01-20 17:06:28.692 INFO nova.compute.claims [req-109ab176-e798-4046-9a79-6983845772bd admin admin] [instance: 3aebf982-09f0-4b6a-b763-9ce594277963] attempting claim: memory 64 MB, disk 0 GB
2016-01-20 17:06:28.693 INFO nova.compute.claims [req-109ab176-e798-4046-9a79-6983845772bd admin admin] [instance: 3aebf982-09f0-4b6a-b763-9ce594277963] total memory: 10000 MB, used: 204.00 MB
2016-01-20 17:06:28.693 INFO nova.compute.claims [req-109ab176-e798-4046-9a79-6983845772bd admin admin] [instance: 3aebf982-09f0-4b6a-b763-9ce594277963] memory limit: 15000.00 MB, free: 14296.00 MB
2016-01-20 17:06:28.694 INFO nova.compute.claims [req-109ab176-e798-4046-9a79-6983845772bd admin admin] [instance: 3aebf982-09f0-4b6a-b763-9ce594277963] total disk: 9 GB, used: 0.00 GB
2016-01-20 17:06:28.694 INFO nova.compute.claims [req-109ab176-e798-4046-9a79-6983845772bd admin admin] [instance: 3aebf982-09f0-4b6a-b763-9ce594277963] disk limit: 9.00 GB, free: 9.00 GB
2016-01-20 17:06:28.708 DEBUG nova.compute.resources.vcpu [req-109ab176-e798-4046-9a79-6983845772bd admin admin] total vcpus: 6 vcpus, used: 3.00 vcpus test /opt/stack/nova/nova/compute/resources/vcpu.py:52
2016-01-20 17:06:28.709 DEBUG nova.compute.resources.vcpu [req-109ab176-e798-4046-9a79-6983845772bd admin admin] vcpus limit not specified, defaulting to unlimited test /opt/stack/nova/nova/compute/resources/vcpu.py:56
2016-01-20 17:06:28.709 INFO nova.compute.claims [req-109ab176-e798-4046-9a79-6983845772bd admin admin] [instance: 3aebf982-09f0-4b6a-b763-9ce594277963] claim successful
```

图 7-184

(2) 使用共享存储上的镜像文件，如图 7-185 所示。

```
2016-01-20 17:06:28.860 DEBUG nova.virt.libvirt.driver [req-109ab176-e798-4046-9a79-6983845772bd admin admin] [instance: 3aebf982-09f0-4b6a-b763-9ce594277963] checking instance files accessibility /opt/stack/data/nova/instances/3aebf982-09f0-4b6a-b763-9ce594277963 instance_on_disk /opt/stack/nova/nova/virt/libvirt/driver.py:7081
2016-01-20 17:06:28.860 INFO nova.compute.manager [req-109ab176-e798-4046-9a79-6983845772bd admin admin] disk on shared storage, recreating using existing disk...
```

图 7-185

(3) 启动 instance，如图 7-186 所示。

```
2016-01-20 17:06:30.508 DEBUG nova.virt.libvirt.driver [req-109ab176-e798-4046-9a79-6983845772bd admin admin] [instance: 3aebf982-09f0-4b6a-b763-9ce594277963] Instance is running spawn /opt/stack/nova/nova/virt/libvirt/driver.py:2445
2016-01-20 17:06:30.509 DEBUG nova.virt.driver [req-e3f82c49-be12-4a4d-8a6a-6dd39dc7bc None None] Emitting event <LifecycleEvent: 1453280790.51, 3aebf982-09f0-4b6a-b763-9ce594277963 => Started> emit_event /opt/stack/nova/nova/virt/driver.py:1303
2016-01-20 17:06:30.510 INFO nova.compute.manager [req-e3f82c49-be12-4a4d-8a6a-6dd39dc7bc None None] [instance: 3aebf982-09f0-4b6a-b763-9ce594277963] VM Started (Lifecycle Event)
2016-01-20 17:06:30.516 2710 INFO nova.virt.libvirt.driver [-] [instance: 3aebf982-09f0-4b6a-b763-9ce594277963] Instance spawned successfully.
```

图 7-186

Evacuate 操作完成后, instance 在 devstack-controller 上运行。

### 7.3.19 Instance 操作总结

前面我们讨论了 Instance 的若干操作, 有的操作功能比较类似, 也有各自的适用场景, 现在是时候系统地总结一下了, 如图 7-187 所示。

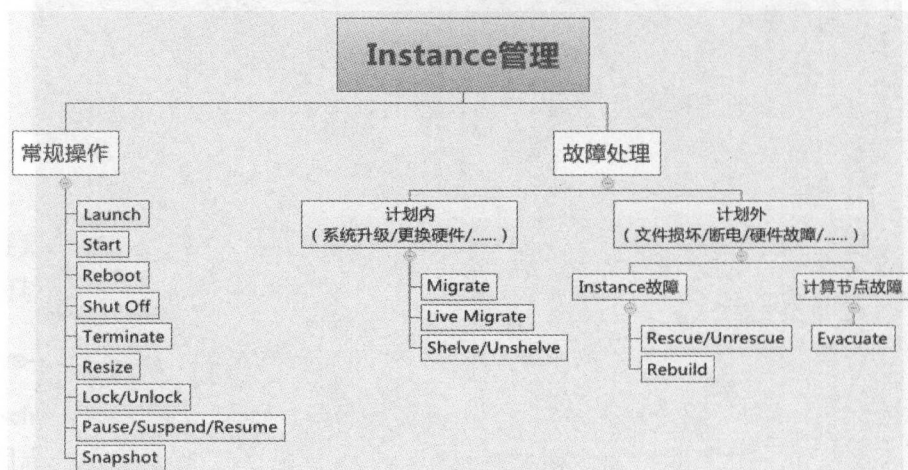


图 7-187

如图 7-187 所示, 我们把对 Instance 的管理按运维工作的场景分为两类: 常规操作和故障处理。

#### 1. 常规操作

常规操作中, Launch、Start、Reboot、Shut Off 和 Terminate 都很好理解。

下面几个操作重点回顾一下:

##### ● Resize

通过应用不同的 flavor 调整分配给 instance 的资源。

##### ● Lock/Unlock

可以防止对 instance 的误操作。

##### ● Pause/Suspend/Resume

暂停当前 instance, 并在以后恢复。

Pause 和 Suspend 的区别在于 Pause 将 instance 的运行状态保存在计算节点的内存中, 而 Suspend 保存在磁盘上。

Pause 的优点是 Resume 的速度比 Suspend 快; 缺点是如果计算节点因某种原因重启, 内存数据丢失, 就无法 Resume 了, 而 Suspend 则没有这个问题。



- Snapshot

备份 instance 到 Glance。

Snapshot 生成的 image 可用于故障恢复，或者以此为模板部署新的 instance。

## 2. 故障处理

故障处理有两种场景：计划内和计划外。

计划内是指提前安排时间窗口做的维护工作，比如服务器定期微码升级，添加更换硬件等。

计划外是指发生了没有预料到的突发故障，比如强行关机造成 OS 系统文件损坏，服务器掉电，硬件故障等。

### (1) 计划内故障处理

对于计划内的故障处理，可以在维护窗口中将 instance 迁移到其他计算节点。

涉及如下操作：

- Migrate

将 instance 迁移到其他计算节点。

迁移之前，instance 会被 Shut Off，支持共享存储和非共享存储。

- Live Migrate

与 Migrate 不同，Live Migrate 能不停机在线地迁移 instance，保证了业务的连续性。也支持共享存储和非共享存储（Block Migration）

- Shelve/Unshelve

Shelve 将 instance 保存到 Glance 上，之后可通过 Unshelve 重新部署。

Shelve 操作成功后，instance 会从原来的计算节点上删除。

Unshelve 会重新选择节点部署，可能不是原节点。

### (2) 计划外故障处理

计划外的故障按照影响的范围又分为两类：Instance 故障和计算节点故障

#### ① Instance 故障

Instance 故障只限于某一个 instance 的操作系统层面，系统无法正常启动。

可以使用如下操作修复 instance：

- Rescue/Unrescue

用指定的启动盘启动，进入 Rescue 模式，修复受损的系统盘。成功修复后，通过 Unrescue 正常启动 instance。

- Rebuild

如果 Rescue 无法修复，则只能通过 Rebuild 从已有的备份恢复。Instance 的备份是通过 snapshot 创建的，所以需要备份策略定期备份。

- ② 计算节点故障

Instance 故障的影响范围局限在特定的 instance，计算节点本身是正常工作的。如果计算节点发生故障，OpenStack 则无法与节点的 nova-compute 通信，其上运行的所有 instance 都会受到影响。这个时候，只能通过 Evacuate 操作在其他正常节点上重建 Instance。

- Evacuate

利用共享存储上 Instance 的镜像文件在其他计算节点上重建 Instance。  
所以提前规划共享存储是关键。

## 7.4 小节

到这里，我们已经学习了 OpenStack Nova 的架构，讨论了 Nova API、Scheduler、Compute 等重要组件，并通过案例详尽地剖析了 Nova 各个操作，最后用一张图总结了这些操作的用途和使用场景。

Nova 是 OpenStack 最重要的项目，处于 OpenStack 的中心。

其他 Keystone、Glance、Cinder 和 Neutron 项目都是为 Nova 其服务的，一定要好好理解。

## 第 8 章

# Block Storage Service ——Cinder

本章我们学习 OpenStack 的 Block Storage Service——Cinder。Cinder 作为 OpenStack 的块存储服务，为 instance 提供虚拟磁盘。

## 8.1 理解 Block Storage

操作系统获得存储空间的方式一般有两种：

- 通过某种协议（SAS、SCSI、SAN、iSCSI 等）挂接裸硬盘，然后分区、格式化、创建文件系统，或者直接使用裸硬盘存储数据（数据库）。
- 通过 NFS、CIFS 等协议，mount 远程的文件系统。

第一种裸硬盘的方式叫做 Block Storage（块存储），每个裸硬盘通常也称作 Volume（卷）

第二种叫做文件系统存储。NAS 和 NFS 服务器，以及各种分布式文件系统提供的都是这种存储。

## 8.2 理解 Block Storage Service

Block Storage Service 提供对 volume 从创建到删除整个生命周期的管理。从 instance 的角度看，挂载的每一个 volume 都是一块硬盘。

OpenStack 提供 Block Storage Service 的是 Cinder，其具体功能是：

- 提供 REST API 使用户能够查询和管理 volume、volume snapshot 以及 volume type。
- 提供 scheduler 调度 volume 创建请求，合理优化存储资源的分配。
- 通过 driver 架构支持多种 back-end（后端）存储方式，包括 LVM，NFS，Ceph 和其他诸如 EMC、IBM 等商业存储产品和方案。

## 8.2.1 Cinder 架构

如图 8-1 所示是 cinder 的逻辑架构图。

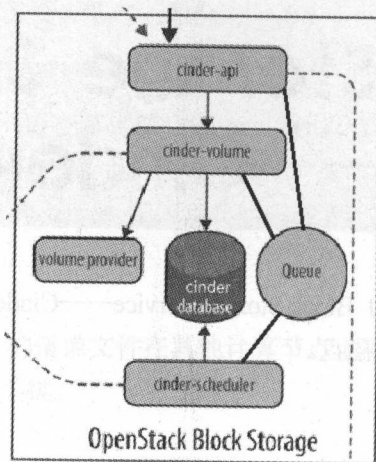


图 8-1

Cinder 包含如下几个组件：

- cinder-api

接收 API 请求，调用 cinder-volume 执行操作。

- cinder-volume

管理 volume 的服务，与 volume provider 协调工作，管理 volume 的生命周期。运行 cinder-volume 服务的节点被称之为存储节点。

- cinder-scheduler

scheduler 通过调度算法选择最合适的存储节点创建 volume。

- volume provider

数据的存储设备，为 volume 提供物理存储空间。

cinder-volume 支持多种 volume provider，每种 volume provider 通过自己的 driver 与 cinder-volume 协调工作。

- Message Queue

Cinder 各个子服务通过消息队列实现进程间通信和相互协作。因为有了消息队列，子服务之间实现了解耦，这种松散的结构也是分布式系统的重要特征。



## ● Database

Cinder 有一些数据需要存放到数据库中，一般使用 MySQL。数据库是安装在控制节点上的，比如在我们的实验环境中，可以访问名称为“cinder”的数据库。如图 8-2 所示。

```

root@devstack-controller:~# su - stack
stack@devstack-controller:~$ mysql
Welcome to the MySQL monitor.  Commands end with \n.
Your MySQL connection id is 12946
Server version: 5.5.46-0ubuntu0.14.04.2 (Ubuntu)

Copyright (c) 2000, 2015, Oracle and/or its affiliates.
Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h;' for help. Type '\c;' to clear the
current input statement.

mysql> use cinder;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with 'SHOW
DATABASES;'.
Database changed
mysql> show tables;
+-----+
Tables_in_cinder
+-----+
backups
cgsnapshots
consistencygroups
driver_initiator_data
encryption
image_volume_cache_entries
iscsi_targets
migrate_version

```

图 8-2

## 8.2.2 物理部署方案

Cinder 的服务会部署在两类节点上，控制节点和存储节点。我们来看看控制节点 devstack-controller 上都运行了哪些 cinder-\* 子服务，如图 8-3 所示。

```

root@devstack-controller:~# ps -e | grep cinder
2868 pts/22    10:02:06 cinder-scheduler
2881 pts/21     05:14:12 cinder-api
2884 pts/23     05:12:46 cinder-volume

```

图 8-3

cinder-api 和 cinder-scheduler 部署在控制节点上，这个很合理。

至于 cinder-volume 也在控制节点上可能有些同学就会迷糊了：cinder-volume 不是应该部署在存储节点上吗？

要回答这个问题，首先要搞清楚一个事实：

OpenStack 是分布式系统，其每个子服务都可以部署在任何地方，只要网络能够连通。

无论是哪个节点，只要上面运行了 cinder-volume，它就是一个存储节点，当然，该节点上也可以运行其他 OpenStack 服务。

cinder-volume 是一顶存储节点帽子，cinder-api 是一顶控制节点帽子。在我们的环境中，devstack-controller 同时戴上了这两顶帽子，所以它既是控制节点，又是存储节点。当然，我们也可以用专门的节点来运行 cinder-volume。

这再一次展示了 OpenStack 分布式架构部署上的灵活性：可以将所有服务都放在一台物理

机上，用作一个 All-in-One 的测试环境；而在生产环境中可以将服务部署在多台物理机上，获得更好的性能和高可用。

RabbitMQ 和 MySQL 通常部署在控制节点上。

另外，也可以用 `cinder service list` 查看 `cinder-*` 子服务都分布在哪些节点上，如图 8-4 所示。

```
root@devstack-controller:~# source /opt/stack/devstack/openrc admin admin
root@devstack-controller:~# cinder service list
```

Binary	Host	Zone	Status
cinder-scheduler	devstack-controller	nova	enabled
cinder-volume	devstack-controller@lvmdriver-1	nova	enabled

```
root@devstack-controller:~#
```

图 8-4

还有一个问题：volume provider 放在那里？

一般来讲，volume provider 是独立的。cinder-volume 使用 driver 与 volume provider 通信并协调工作。所以只需要将 driver 与 cinder-volume 放到一起就可以了。在 cinder-volume 的源代码目录下有很多 driver，支持不同的 volume provider，如图 8-5 所示。

```
root@devstack-controller:~# ls -l /opt/stack/cinder/cinder/volume/drivers
total 780
-rw-r--r-- 1 stack stack 837 Dec 10 20:17 __init__.py
-rw-r--r-- 1 stack stack 375 Dec 10 20:47 __init__.pyc
-rw-r--r-- 1 stack stack 8666 Dec 10 20:17 block_device.py
-rw-r--r-- 1 stack stack 21824 Dec 10 20:17 blockbridge.py
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17
-rw-r--r-- 1 stack stack 16592 Dec 10 20:17 datara.py
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17
-rw-r--r-- 1 stack stack 19937 Dec 10 20:17 drbdmanagedrv.py
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17
-rw-r--r-- 1 stack stack 23543 Dec 10 20:17 eglx.py
-rw-r--r-- 1 stack stack 17430 Dec 10 20:17 glusterfs.py
-rw-r--r-- 1 stack stack 25914 Dec 10 20:17 hgst.py
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17
drwxr-xr-x 3 stack stack 4096 Dec 10 20:17
drwxr-xr-x 3 stack stack 4096 Dec 10 20:17
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17
-rw-r--r-- 1 stack stack 33306 Dec 10 20:17 lvm.py
-rw-r--r-- 1 stack stack 25302 Dec 10 20:47 lvm.pyc
drwxr-xr-x 4 stack stack 4096 Dec 10 20:17
-rw-r--r-- 1 stack stack 19460 Dec 10 20:17 nfs.py
-rw-r--r-- 1 stack stack 39319 Dec 10 20:17 nimble.py
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17
-rw-r--r-- 1 stack stack 41338 Dec 10 20:17 pure.py
-rw-r--r-- 1 stack stack 17012 Dec 10 20:17 quobyte.py
-rw-r--r-- 1 stack stack 44375 Dec 10 20:17 rbd.py
-rw-r--r-- 1 stack stack 56633 Dec 10 20:17 remotefs.py
drwxr-xr-x 3 stack stack 4096 Dec 10 20:17
```

图 8-5

后面我们会以 LVM 和 NFS 这两种 volume provider 为例讨论 cinder-volume 的使用，其他 volume provider 可以查看 OpenStack 的 configuration 文档。

### 8.2.3 从 volume 创建流程看 cinder-\*子服务如何协同工作

对于 Cinder 学习来说，Volume 创建是一个非常好的场景，涉及各个 cinder-\* 子服务，如图 8-6 所示是流程图。

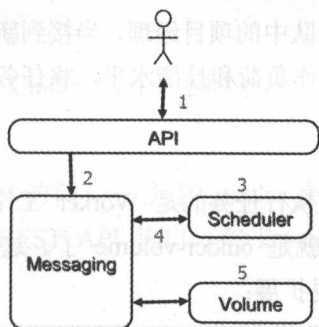


图 8-6

- 客户（可以是 OpenStack 最终用户，也可以是其他程序）向 API（cinder-api）发送请求：“帮我创建一个 volume”。
- API 对请求做一些必要处理后，向 Messaging（RabbitMQ）发送了一条消息：“让 Scheduler 创建一个 volume”。
- Scheduler（cinder-scheduler）从 Messaging 获取到 API 发给它的消息，然后执行调度算法，从若干存储点中选出节点 A。
- Scheduler 向 Messaging 发送了一条消息：“让存储节点 A 创建这个 volume”。
- 存储节点 A 的 Volume（cinder-volume）从 Messaging 中获取到 Scheduler 发给它的消息，然后通过 driver 在 volume provider 上创建 volume。

上面是创建虚拟机最核心的几个步骤，当然省略了很多细节，我们会在后面的章节详细讨论。

## 8.2.4 Cinder 的设计思想

Cinder 延续了 Nova 以及其他组件的设计思想。

### 1. API 前端服务

cinder-api 作为 Cinder 组件对外的唯一窗口，向客户暴露 Cinder 能够提供的功能，当客户需要执行 volume 相关的操作，能且只能向 cinder-api 发送 REST 请求。这里的客户包括终端用户、命令行和 OpenStack 其他组件。

设计 API 前端服务的好处在于：

- 对外提供统一接口，隐藏实现细节。
- API 提供 REST 标准调用服务，便于与第三方系统集成。
- 可以通过运行多个 API 服务实例轻松实现 API 的高可用，比如运行多个 cinder-api 进程。

### 2. Scheduler 调度服务

Cinder 可以有多个存储节点，当需要创建 volume 时，cinder-scheduler 会根据存储节点的属性和资源使用情况选择一个最合适的节点来创建 volume。

调度服务就好比是一个开发团队中的项目经理，当接到新的开发任务时，项目经理会根据任务的难度、每个团队成员目前的工作负荷和技能水平，将任务分配给最合适的开发人员。

3. Worker 工作服务

调度服务只管分配任务，真正执行任务的是 Worker 工作服务。

在 Cinder 中，这个 Worker 就是 cinder-volume 了。这种 Scheduler 和 Worker 之间职能上的划分使得 OpenStack 非常容易扩展：

当存储资源不够时可以增加存储节点（增加 Worker）。

当客户的请求量太大调度不过来时，可以增加 Scheduler。

4. Driver 框架

OpenStack 作为开放的 Infrastructure as a Service 云操作系统，支持业界各种优秀的技术，这些技术可能是开源免费的，也可能是商业收费的。

这种开放的架构使得 OpenStack 保持技术上的先进性，具有很强的竞争力，同时又不会造成厂商锁定（Lock-in）。

那 OpenStack 的这种开放性体现在哪里呢？一个重要的方面就是采用基于 Driver 的框架。

以 Cinder 为例，存储节点支持多种 volume provider，包括 LVM，NFS，Ceph，GlusterFS，以及 EMC，IBM 等商业存储系统。

cinder-volume 为这些 volume provider 定义了统一的 driver 接口，volume provider 只需要实现这些接口，就可以 driver 的形式即插即用到 OpenStack 中，如图 8-7 所示是 cinder driver 的架构示意图。

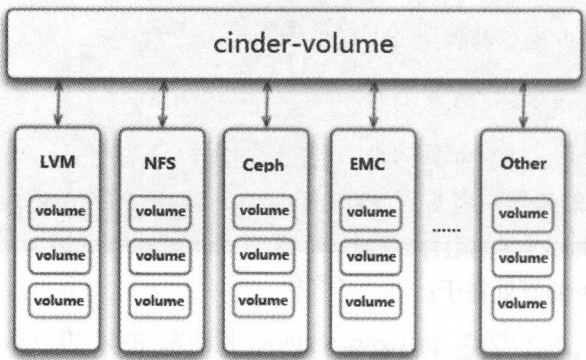


图 8-7

在 cinder-volume 的配置文件 /etc/cinder/cinder.conf 中，volume\_driver 配置项设置该存储节点使用哪种 volume provider 的 driver，图 8-8 的示例表示使用的是 LVM。

```
volume_driver = cinder.volume.drivers.lvm.LVMVolumeDriver
```

图 8-8



## 8.2.5 Cinder 组件详解

从本节开始，我们将详细讲解 Cinder 的各个子服务。

### 1. cinder-api

cinder-api 是整个 Cinder 组件的门户，所有 cinder 的请求都首先由 nova-api 处理。cinder-api 向外界暴露若干 HTTP REST API 接口。在 keystone 中我们可以查询 cinder-api 的 endpoints。如图 8-9 所示。

```
root@devstack-controller:~# source /opt/stack/devstack/openrc admin admin
root@devstack-controller:~# openstack endpoint show cinder
```

Field	Value
adminurl	http://192.168.104.10:8776/v1/\$(tenant_id)s
enabled	True
id	a9545ea0d6c04fb7a48ee91c1b897aa5
internalurl	http://192.168.104.10:8776/v1/\$(tenant_id)s
publicurl	http://192.168.104.10:8776/v1/\$(tenant_id)s
region	RegionOne
service_id	80fb9a8292a74823a5c7d2097ef13081
service_name	cinder
service_type	volume

图 8-9

客户端可以将请求发送到 endpoints 指定的地址，向 cinder-api 请求操作。

当然，作为最终用户的我们不会直接发送 Rest API 请求。OpenStack CLI, Dashboard 和其他需要跟 Cinder 交换的组件会使用这些 API。

cinder-api 对接收到的 HTTP API 请求会做如下处理：

- 检查客户端传入的参数是否合法有效。
- 调用 cinder 其他子服务的处理客户端请求。
- 将 cinder 其他子服务返回的结果序列号并返回给客户端。

cinder-api 接受哪些请求呢？简单地说，只要是 Volume 生命周期相关的操作，cinder-api 都可以响应。大部分操作都可以在 Dashboard 上看到。

打开 Volume 管理界面，如图 8-10 所示。

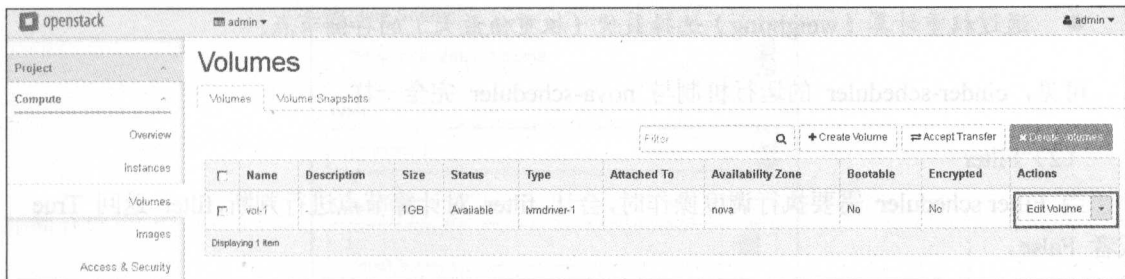


图 8-10

单击 Edit Volume 右侧的下拉箭头，列表中就是 cinder-api 可执行的操作，如图 8-11 所示。

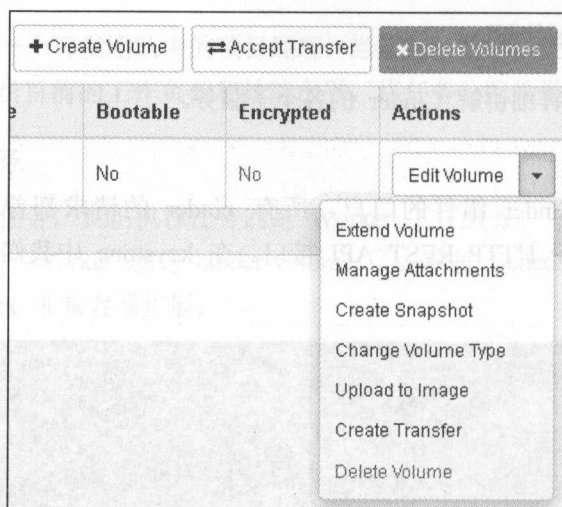


图 8-11

## 2. cinder-scheduler

创建 Volume 时, cinder-scheduler 会基于容量、Volume Type 等条件选择出最合适的存储节点, 然后让其创建 Volume。

下面介绍 cinder-scheduler 是如何实现这个调度工作的。

在 `/etc/cinder/cinder.conf` 中, cinder 通过 `schedulerdriver`, `schedulerdefault_filters` 和 `schedulerdefaultweighers` 三个参数来配置 cinder-scheduler。

### (1) Filter scheduler

Filter scheduler 是 cinder-scheduler 默认的调度器。

```
scheduler_driver=cinder.scheduler.filter_scheduler.FilterScheduler
```

与 Nova 一样, Cinder 也允许使用第三方 scheduler, 配置 `scheduler_driver` 即可。

scheduler 调度过程如下:

- 通过过滤器 (filter) 选择满足条件的存储节点 (运行 cinder-volume)。
- 通过权重计算 (weighting) 选择最优 (权重值最大) 的存储节点。

可见, cinder-scheduler 的运行机制与 nova-scheduler 完全一样。

### (2) Filter

当 Filter scheduler 需要执行调度操作时, 会让 filter 对计算节点进行判断, filter 返回 True 或者 False。

cinder.conf 中 `schedulerdefaultfilters` 选项指定 filter scheduler 使用的 filter, 默认值如下:

```
scheduler_default_filters = AvailabilityZoneFilter, CapacityFilter, CapabilitiesFilter
```

Filter scheduler 将按照列表中的顺序依次过滤。

### ① AvailabilityZoneFilter

为提高容灾性和提供隔离服务，可以将存储节点和计算节点划分到不同的 Availability Zone 中。例如把一个机架上的机器划分到一个 Availability Zone 中。OpenStack 默认有一个命名为“Nova”Availability 的 Zone，所有的节点初始都是放在“Nova”中的。用户可以根据需要创建自己的 Availability Zone，如图 8-12 所示。

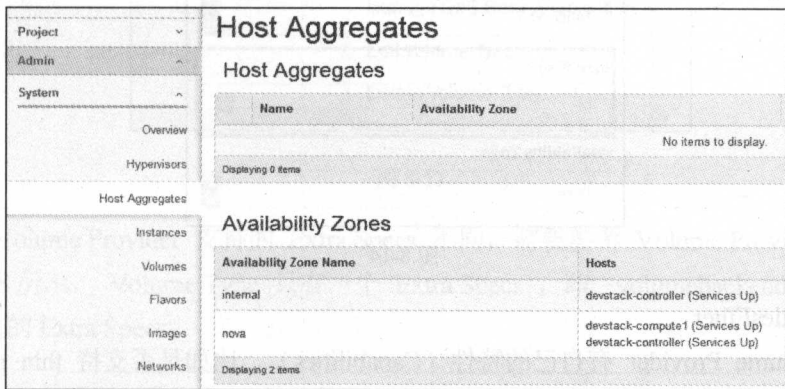


图 8-12

创建 Volume 时，需要指定 Volume 所属的 Availability Zone，如图 8-13 所示。

The screenshot shows the 'Create Volume' form. It has the following fields: 'Volume Name' (text input), 'Description' (text area), 'Volume Source' (dropdown menu with 'No source, empty volume' selected), 'Type' (dropdown menu with 'lvmdriver-1' selected), 'Size (GB) \*' (text input with '1' and a unit selector), and 'Availability Zone' (dropdown menu with 'nova' selected). The 'Availability Zone' field is highlighted with a red box.

图 8-13

cinder-scheduler 在做 filtering 时, 会使用 AvailabilityZoneFilter 将不属于指定 Availability Zone 的存储节点过滤掉。

### ② CapacityFilter

创建 Volume 时, 用户会指定 Volume 的大小。CapacityFilter 的作用是将存储空间不能满足 Volume 创建需求的存储节点过滤掉, 如图 8-14 所示。

The form contains three fields:

- Type:** A dropdown menu with the value 'lvmdriver-1' selected.
- Size (GB) \*:** A text input field with the value '1'.
- Availability Zone:** A dropdown menu with the value 'nova' selected.

图 8-14

### ③ CapabilitiesFilter

不同的 Volume Provider 有自己的特性 (Capabilities), 比如是否支持 thin provision 等。Cinder 允许用户创建 Volume 时通过 Volume Type 指定需要的 Capabilities, 如图 8-15 所示。

The form contains three fields:

- Volume Source:** A dropdown menu with the value 'No source, empty volume' selected.
- Type:** A dropdown menu with the value 'lvmdriver-1' selected.
- Size (GB) \*:** A text input field with the value '1'.

图 8-15

Volume Type 可以根据需要定义若干 Capabilities, 详细描述 Volume 的属性。Volume Type 的作用与 Nova 的 flavor 类似。

Volume Type 在 Admin → System → Volume 菜单里管理, 如图 8-16 所示。

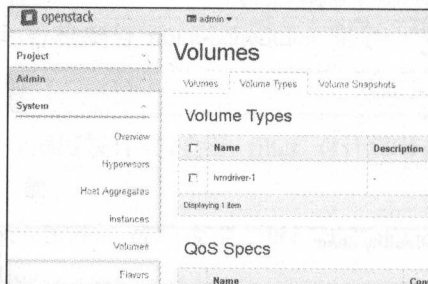


图 8-16



通过 Volume Type 的 Extra Specs 定义 Capabilities。Extra Specs 是用 Key-Value 的形式定义，如图 8-17 所示。

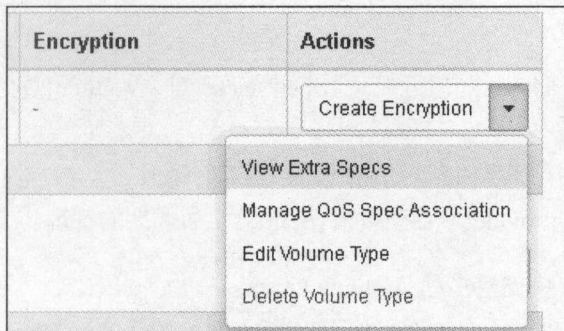


图 8-17

不同的 Volume Provider 支持的 Extra Specs 不同，需要参考 Volume Provider 的文档。

如图 8-18 所示，Volume Type 只有一个 Extra Specs，即“volumebackendname”，这是最重要也是必需的 Extra Specs。

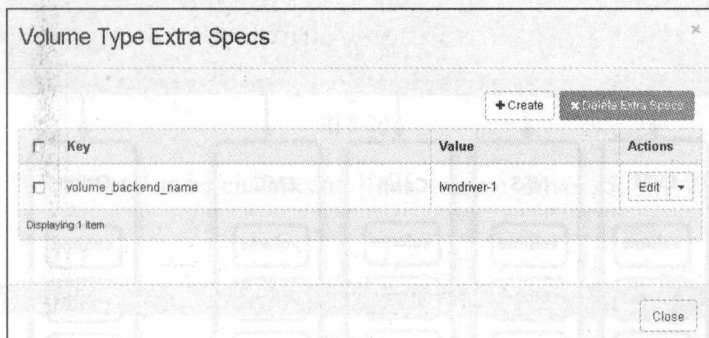


图 8-18

cinder-volume 会在自己的配置文件 `/etc/cinder/cinder.conf` 中，设置“volume\_backend\_name”参数，其作用是为存储节点的 Volume Provider 命名。这样，CapabilitiesFilter 就可以通过 Volume Type 的“volume\_backend\_name”筛选出指定的 Volume Provider。

不同的存储节点可以在各自的 `cinder.conf` 中配置相同的 volumebackendname，这是允许的。因为虽然存储节点不同，但它们可能使用的是一种 Volume Provider。

如果在第一步 filtering 环节选出了多个存储节点，那么接下来的 weighting 环节会挑选出最合适的一个节点。

### (3) Weighter

Filter Scheduler 通过 `scheduler_default_weighters` 指定计算权重的 weighter，默认为 CapacityWeighter。

```
scheduler_default_weighters = CapacityWeigher
```

如命名所示，CapacityWeigher 基于存储节点的空闲容量计算权重值，空闲最多的胜出。

3. cinder-volume

cinder-volume 在存储节点上运行，OpenStack 对 Volume 的操作，最后都是交给 cinder-volume 来完成的。

cinder-volume 自身并不管理真正的存储设备，存储设备是由 volume provider 管理的。cinder-volume 与 volume provider 一起实现 volume 生命周期的管理。

(1) 通过 Driver 架构支持多种 Volume Provider

接着的问题是：现在市面上有这么多块存储产品和方案（volume provider），cinder-volume 如何与它们配合呢？这就是我们之前讨论过的 Driver 架构。

cinder-volume 为这些 volume provider 定义了统一的接口，volume provider 只需要实现这些接口，就可以 Driver 的形式即插即用 OpenStack 系统中，如图 8-19 所示是 Cinder Driver 的架构示意图。

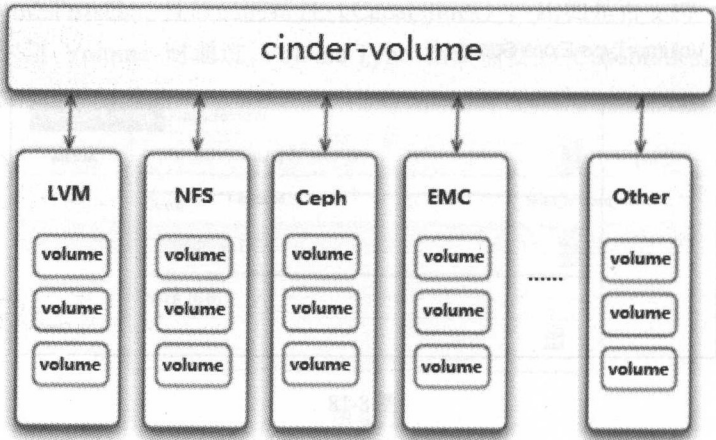


图 8-19

我们可以在 /opt/stack/cinder/cinder/volume/drivers/ 目录下查看到 OpenStack 源代码中已经自帶了很多 volume provider 的 Driver，如图 8-20 所示。

```

root@devstack-controller:~# ls -l /opt/stack/cinder/cinder/volume/drivers/
total 780
-rw-r--r-- 1 stack stack 837 Dec 10 20:17 __init__.py
-rw-r--r-- 1 stack stack 375 Dec 10 20:47 __init__.pyc
-rw-r--r-- 1 stack stack 8666 Dec 10 20:17 block_device.py
-rw-r--r-- 1 stack stack 21824 Dec 10 20:17 blockbridge.py
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17
-rw-r--r-- 1 stack stack 16592 Dec 10 20:17 datara.py
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17
-rw-r--r-- 1 stack stack 19937 Dec 10 20:17 drbdmanagedrv.py
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17
-rw-r--r-- 1 stack stack 23543 Dec 10 20:17 eglx.py
-rw-r--r-- 1 stack stack 17430 Dec 10 20:17 glusterfs.py
-rw-r--r-- 1 stack stack 25914 Dec 10 20:17 hgst.py
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17
drwxr-xr-x 3 stack stack 4096 Dec 10 20:17
drwxr-xr-x 3 stack stack 4096 Dec 10 20:17
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17
-rw-r--r-- 1 stack stack 33306 Dec 10 20:17 lvm.py
-rw-r--r-- 1 stack stack 25302 Dec 10 20:47 lvm.pyc
drwxr-xr-x 4 stack stack 4096 Dec 10 20:17
-rw-r--r-- 1 stack stack 19460 Dec 10 20:17 nfs.py
-rw-r--r-- 1 stack stack 39319 Dec 10 20:17 nimble.py
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17
-rw-r--r-- 1 stack stack 41338 Dec 10 20:17 pure.py
-rw-r--r-- 1 stack stack 17012 Dec 10 20:17 quobyte.py
-rw-r--r-- 1 stack stack 44375 Dec 10 20:17 rbd.py
-rw-r--r-- 1 stack stack 56633 Dec 10 20:17 remotefs.py
drwxr-xr-x 3 stack stack 4096 Dec 10 20:17
-rw-r--r-- 1 stack stack 11536 Dec 10 20:17 scalability.py
-rw-r--r-- 1 stack stack 20782 Dec 10 20:17 sheepdog.py
-rw-r--r-- 1 stack stack 23340 Dec 10 20:17 smbfs.py
-rw-r--r-- 1 stack stack 55037 Dec 10 20:17 solidfire.py
-rw-r--r-- 1 stack stack 33438 Dec 10 20:17 srb.py
-rw-r--r-- 1 stack stack 34581 Dec 10 20:17 tintri.py
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17
-rw-r--r-- 1 stack stack 12987 Dec 10 20:17 vzstorage.py
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17
-rw-r--r-- 1 stack stack 69116 Dec 10 20:17 xio.py
drwxr-xr-x 2 stack stack 4096 Dec 10 20:17

```

图 8-20

存储节点在配置文件 `/etc/cinder/cinder.conf` 中用 `volume_driver` 选项配置使用的 driver，如图 8-21 所示。

```
volume_driver = cinder.volume.drivers.lvm.LVMvolumeDriver
```

图 8-21

这里 LVM 是我们使用的 volume provider。

## (2) 定期向 OpenStack 报告计算节点的状态

在前面 `cinder-scheduler` 会用到 `CapacityFilter` 和 `CapacityWeigher`，它们都是通过存储节点的空闲容量来做筛选。那这里有个问题：Cinder 是如何得知每个存储节点的空闲容量信息的呢？答案就是：`cinder-volume` 会定期向 Cinder 报告。

从 `cinder-volume` 的日志 `/opt/stack/logs/c-vol.log` 可以发现每隔一段时间，`cinder-volume` 就会报告当前存储节点的资源使用情况，如图 8-22 所示。

```

2016-01-26 19:14:12.170 DEBUG cinder.manager [req-a596207a-3be8-4599-9d0f-6d204062818c None] Notifying schedulers of capabilities .....publish_service_capabilities /opt/stack/cinder/cinder/manager.py:156
2016-01-26 19:14:12.172 DEBUG oslo.service.periodic_task [req-a596207a-3be8-4599-9d0f-6d204062818c None] Running periodic task volumeManager._report_driver_status run_periodic_tasks /usr/local/lib/python2.7/dist-packages/oslo_service/periodic_task.py:213
2016-01-26 19:14:12.173 DEBUG cinder.volume.drivers.lvm [req-a596207a-3be8-4599-9d0f-6d204062818c None] updating volume stats _update_volume_stats /opt/stack/cinder/cinder/volume/drivers/lvm.py:175
2016-01-26 19:14:12.173 DEBUG oslo_concurrency.processutils [req-a596207a-3be8-4599-9d0f-6d204062818c None] Running cmd (subprocess): env LC_ALL=C vgs --noheadings --unit=g -o name,size,free,lvs_count,uuid --separator ; --nosuffix stack-volumes-lvmdriver-1 execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-01-26 19:14:12.191 DEBUG oslo_concurrency.processutils [req-a596207a-3be8-4599-9d0f-6d204062818c None] CMD "env LC_ALL=C vgs --noheadings --unit=g -o name,size,free,lvs_count,uuid --separator ; --nosuffix stack-volumes-lvmdriver-1" returned: 0 in 0.018s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
2016-01-26 19:14:12.192 DEBUG oslo_concurrency.processutils [req-a596207a-3be8-4599-9d0f-6d204062818c None] Running cmd (subprocess): env LC_ALL=C lvs --noheadings --unit=g -o vg_name,name,size --nosuffix stack-volumes-lvmdriver-1 execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-01-26 19:14:12.210 DEBUG oslo_concurrency.processutils [req-a596207a-3be8-4599-9d0f-6d204062818c None] CMD "env LC_ALL=C lvs --noheadings --unit=g -o vg_name,name,size --nosuffix stack-volumes-lvmdriver-1" returned: 0 in 0.018s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280

```

图 8-22

因为在我们的实验环境中存储节点使用的是 LVM，所以在上面的日志看到存储节点通过“vgs”和“lvs”这两个命令获取 LVM 的容量使用信息。

### (3) 实现 volume 生命周期管理

Cinder 对 volume 的生命周期的管理最终都是通过 cinder-volume 完成的，包括 volume 的 create、extend、attach、snapshot、delete 等，后面我们会详细讨论。

## 8.2.6 通过场景学习 Cinder

Volume 从创建到删除的整个生命周期都是由 Cinder 管理。

后面各小节我们将以 volume 生命周期中的不同操作场景为例，详细讨论 Cinder 不同组件之间如何协调工作，并通过日志的分析帮助大家加深对 Cinder 的理解。

### 1. LVM Volume Provider

Cinder 将 LVM 作为默认的 volume provider。Devstack 安装之后，/etc/cinder/cinder 已经配置好了 LVM，如图 8-23 所示。

```

default_volume_type = lvmdriver-1
enabled_backends = lvmdriver-1

[lvmdriver-1]
volume_group = stack-volumes-lvmdriver-1
volume_driver = cinder.volume.drivers.lvm.LVMVolumeDriver
volume_backend_name = lvmdriver-1

```

图 8-23

上面的配置定义了名为“lvmdriver-1”的 volume provider，也称作 back-end。其 driver 是 LVM，LVM 的 volume group 名为“stack-volumes-lvmdriver-1”。

Devstack 安装时并没有自动创建 volume group，所以需要手工创建。



下面我们分步骤演示一下在 `/dev/sdb` 上创建 VG “stack-volumes-lvmdriver-1” 的过程。  
首先创建 physical volume `/dev/sdb`，如图 8-24 所示。

```
root@devstack-controller:~# pvcreate /dev/sdb
Device /dev/sdb not found (or ignored by filtering).
```

图 8-24

Linux 的 lvm 默认配置不允许在 `/dev/sdb` 上创建 PV，需要将 `sdb` 添加到 `/etc/lvm.conf` 的 `filter` 中，如图 8-25 所示。

```
global_filter = [ ]

root@devstack-controller:~# pvcreate /dev/sdb
Physical volume "/dev/sdb" successfully created
```

图 8-25

然后创建 VG `stack-volumes-lvmdriver-1`，如图 8-26 所示。

```
root@devstack-controller:~# vgcreate stack-volumes-lvmdriver-1 /dev/sdb
volume group "stack-volumes-lvmdriver-1" successfully created
```

图 8-26

打开 Web GUI，可以看到 OpenStack 已经创建了 Volume Type “lvmdriver-1”，如图 8-27 所示。

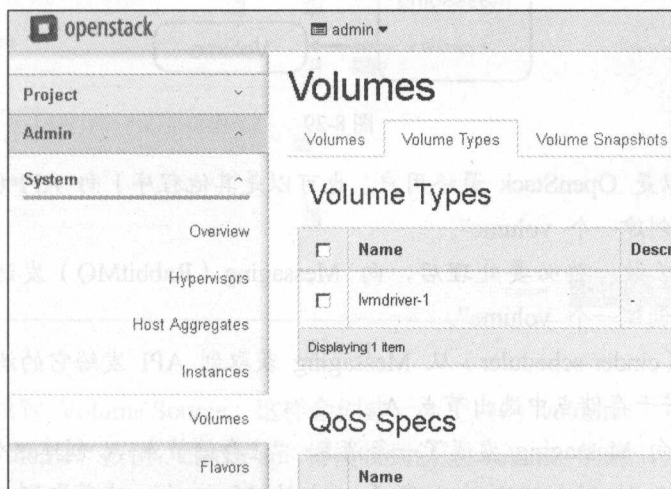


图 8-27

其 Extra Specs `volume_backend_name` 为 `lvmdriver-1`，如图 8-28 所示。

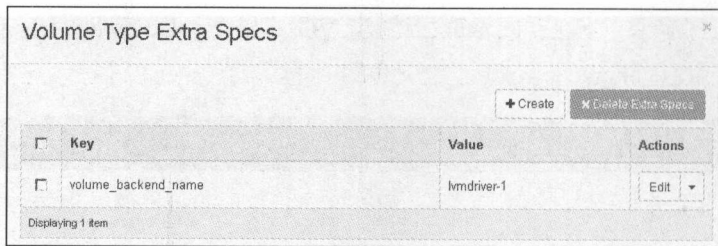


图 8-28

后面各小节都将以 LVM 为 volume provider 详细讨论 volume 的各种操作。

## 2. Create

首先我们来学习 Cinder 是如何创建 volume 的，如图 8-29 所示的流程图。

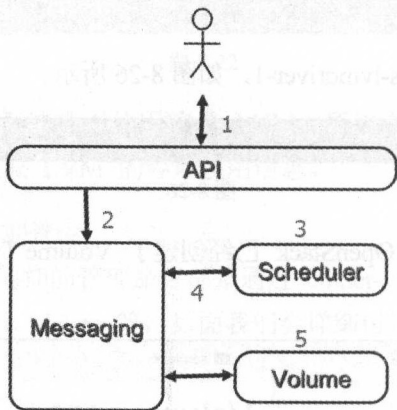


图 8-29

- 客户（可以是 OpenStack 最终用户，也可以是其他程序）向 API（cinder-api）发送请求：“帮我创建一个 volume”。
- API 对请求做一些必要处理后，向 Messaging（RabbitMQ）发送了一条消息：“让 Scheduler 创建一个 volume”。
- Scheduler（cinder-scheduler）从 Messaging 获取到 API 发给它的消息，然后执行调度算法，从若干存储点中选出节点 A。
- Scheduler 向 Messaging 发送了一条消息：“让存储节点 A 创建这个 volume”。
- 存储节点 A 的 Volume（cinder-volume）从 Messaging 中获取到 Scheduler 发给它的消息，然后通过 driver 在 volume provider 上创建 volume。

下面我们详细讨论每一个步骤。

### （1）向 cinder-api 发送请求

客户（可以是 OpenStack 最终用户，也可以是其他程序）向 cinder-api 发送请求：“帮我创建一个 volume。”

GUI 上操作的菜单为 Project → Compute → Volumes → Create Volume，如图 8-30 所示。

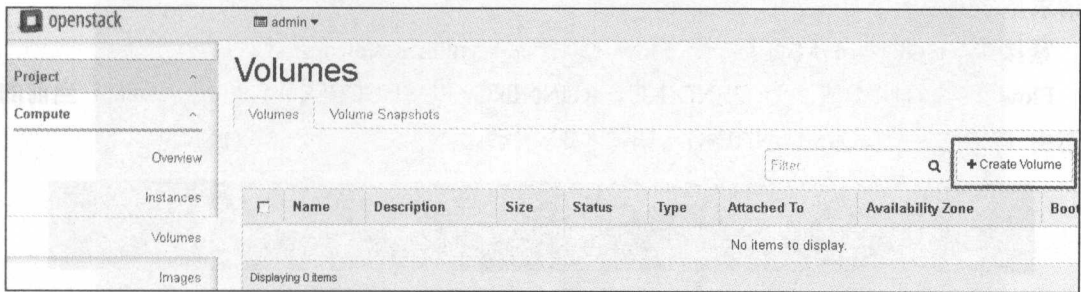


图 8-30

设置 volume 的名称，volume type，大小，Availability Zone 等基本信息，如图 8-31 所示。

图 8-31

这里我们没有设置 Volume Source，这样会创建一个空白的 volume。

单击“Create Volume”按钮，cinder-api 将接收到创建 volume 的请求。

查看 cinder-api 日志 /opt/stack/logs/c-api.log，如图 8-32 所示。

```
2016-01-27 17:07:23.633 INFO cinder.api.openstack.wsgi [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] POST http://192.168.104.10:8776/v2/aa81b851d2a54484b6f3984ab2c5d4e4/volumes
2016-01-27 17:07:23.635 DEBUG cinder.api.v2.volumes [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Create volume request body: {'volume': {'status': 'creating', 'size': 1, 'project_id': None, 'user_id': None, 'description': '', 'imageRef': None, 'availability_zone': 'nova', 'scheduler_hints': {}, 'multiattach': False, 'attach_status': 'detached', 'volume_type': 'lvmdriver-1', 'consistencygroup_id': None, 'source_vol_id': None, 'snapshot_id': None, 'metadata': {}, 'source_replica': None, 'name': 'vol-1'}} create /opt/stack/cinder/cinder/api/v2/volumes.py:319
2016-01-27 17:07:23.647 INFO cinder.api.v2.volumes [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Create volume of 1 GB
```

图 8-32

日志显示 cinder-api 接收到一个 POST 类型的 REST API，经过对 HTTP body 的分析，该请求是：创建一个 1GB 的 volume。

紧接着，cinder-api 启动了一个 Flow（工作流）volumecreateapi。

Flow 的执行状态依次为 PENDING、RUNNING 和 SUCCESS。volumecreateapi 当前的状态由 PENDING 变为 RUNNING，如图 8-33 所示。

```
2016-01-27 17:07:23.668 DEBUG cinder.volume.api [req-38febe3d-27c9-4dbf-b461-aa67f01b242f None] flow volume_create_api (ddfb7238-a039-465c-b7e5-e40ee83155d9) transitioned into state RUNNING from state PENDING, flow_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:140
```

图 8-33

volume\_create\_api 工作流包含若干 Task，每个 Task 完成特定的任务。这些任务依次为 ExtractVolumeRequestTask、QuotaReserveTask、EntryCreateTask、QuotaCommitTask、VolumeCast Task。

Task 的执行状态也会经历 PENDING、RUNNING 和 SUCCESS 三个阶段。

Task 的名称基本上说明了任务的工作内容 前面几个 Task 主要是做一些创建 volume 的准备工作，比如：ExtractVolumeRequestTask 获取 request 信息，如图 8-34 所示。

```
2016-01-27 17:07:23.672 DEBUG cinder.volume.api [req-38febe3d-27c9-4dbf-b461-aa67f01b242f None] Task cinder.volume.flows.api.create_volume.ExtractVolumeRequestTask; volume:create (f5862217-e64a-4544-abe2-527f634f67d1) transitioned into state RUNNING from state PENDING, task_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:189
2016-01-27 17:07:23.673 DEBUG cinder.volume.flows.api.create_volume [req-38febe3d-27c9-4dbf-b461-aa67f01b242f None] validating volume 1 using validate_int _extract_size /opt/stack/cinder/cinder/volume/flows/api/create_volume.py:186
2016-01-27 17:07:23.719 DEBUG cinder.volume.api [req-5a722737-7874-4012-867c-a7227e0fb40b None] Task cinder.volume.flows.api.create_volume.ExtractVolumeRequestTask; volume:create (f5862217-e64a-4544-abe2-527f634f67d1) transitioned into state SUCCESS from state RUNNING with result {'volume_type_id': u'7cd531f0-4fa2-4a2c-9582-4564cb4f54c3', 'availability_zone': u'nova', 'source_volid': None, 'qos_specs': None, 'consistencygroup_id': None, 'snapshot_id': None, 'size': 1, 'source_replicaid': None, 'volume_type': {'name': u'lvmdriver-1', 'qos_specs_id': None, 'deleted': False, 'created_at': datetime.datetime(2015, 12, 10, 12, 47, 4), 'updated_at': None, 'extra_specs': {'volume_backend_name': u'lvmdriver-1', 'is_public': True, 'deleted_at': None, 'id': u'7cd531f0-4fa2-4a2c-9582-4564cb4f54c3', 'description': None}, 'cgsnapshot_id': None, 'encryption_key_id': None}, task_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:178
```

图 8-34

QuotaReserveTask 预留配额，如图 8-35 所示。

```
2016-01-27 17:07:23.720 DEBUG cinder.volume.api [req-5a722737-7874-4012-867c-a7227e0fb40b None] Task cinder.volume.flows.api.create_volume.QuotaReserveTask; volume:create (14e014f9-6ee1-4065-940b-f7adaaf63fca) transitioned into state RUNNING from state PENDING, task_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:189
2016-01-27 17:07:23.773 DEBUG cinder.volume.quota [req-a2561b49-a7ca-45a4-8f4a-09118c813be1 None] created reservations [61b28b9d-7265-43dd-8530-3b4dde935d3e', '3282f834-2083-4d5b-530a398d1161', '2cfd6df0-729f-4b60-a795-248eeae9c24e', 'b314b246-1b72-425b-a645-d47d3794abba'] reserve /opt/stack/cinder/cinder/quota.py:810
2016-01-27 17:07:23.775 DEBUG cinder.volume.api [req-a2561b49-a7ca-45a4-8f4a-09118c813be1 None] Task cinder.volume.flows.api.create_volume.QuotaReserveTask; volume:create (14e014f9-6ee1-4065-940b-f7adaaf63fca) transitioned into state SUCCESS from state RUNNING with result {'reservations': [61b28b9d-7265-43dd-8530-3b4dde935d3e', '3282f834-2083-4d5b-530a398d1161', '2cfd6df0-729f-4b60-a795-248eeae9c24e', 'b314b246-1b72-425b-a645-d47d3794abba']}, task_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:178
```

图 8-35

EntryCreateTask 在数据库中创建 volume 条目，如图 8-36 所示。



```

2016-01-27 17:07:23.776 DEBUG cinder.volume.api [req-a2561b49-a7ca-45a4-8f4a-09118c8
13be1 None] Task 'cinder.volume.flows.api.create_volume.EntryCreateTask;volume:creat
e' (0985f08c-29e3-4541-b266-7fda697df71c) transitioned into state 'RUNNING' from sta
te 'PENDING' task_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listener
s/logging.py:189
2016-01-27 17:07:23.831 DEBUG cinder.volume.api [req-a2561b49-a7ca-45a4-8f4a-09118c8
13be1 None] Task 'cinder.volume.flows.api.create_volume.EntryCreateTask;volume:creat
e' (0985f08c-29e3-4541-b266-7fda697df71c) transitioned into state 'SUCCESS' from sta
te 'RUNNING' with result: {'volume': <cinder.db.sqlalchemy.models.volume.Object at 0
x7fdf47ed7c10>, 'volume_properties': {'status': 'creating', 'volume_type_id': 'u7cd5
31f0-4fa2-4a2c-9582-4564cb4f54c3', 'multiattach': False, 'reservations': ['61b28b9d-
7265-43dd-8530-3b4dde935d3e', '3282f834-2083-4d5b-93ec-530a398d1161', '2cfd6df0-729f
-4b60-a795-248eeae9c24e', 'b314b246-1b72-425b-a645-d47d3794abba'], 'source_valid': N
one, 'volume_metadata': [], 'qos_specs': None, 'consistencygroup_id': None, 'replica
tion_status': 'disabled', 'snapshot_id': None, 'display_name': 'u'vol-1', 'id': '1e7f
6bd7-cell1-4a73-b95e-aabd65a5b188', 'size': 1, 'user_id': 'u'b02f995f9b734384a5662c8d3
0d661e4', 'source_replica_id': None, 'availability_zone': 'u'nova', 'attach_status':
'detached', 'display_description': 'u', 'cgsnapshot_id': None, 'encryption_key_id': N
one, 'volume_admin_metadata': [], 'project_id': 'u'aas1b851d2a54484b6f3984ab2c5d4e4',
'metadata': {}}, 'volume_id': '1e7f6bd7-cell1-4a73-b95e-aabd65a5b188'} task_receiv
er /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:178

```

图 8-36

QuotaCommitTask 确认配额, 如图 8-37 所示。

```

2016-01-27 17:07:23.833 DEBUG cinder.volume.api [req-a2561b49-a7ca-45a4-8f4a-09118c8
13be1 None] Task 'cinder.volume.flows.api.create_volume.QuotaCommitTask;volume:creat
e' (2a8b14e5-64a8-42b2-88a0-519158f2adc7) transitioned into state 'RUNNING' from sta
te 'PENDING' task_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listener
s/logging.py:189
2016-01-27 17:07:23.854 DEBUG cinder.volume.api [req-a2561b49-a7ca-45a4-8f4a-09118c8
13be1 None] Task 'cinder.volume.flows.api.create_volume.QuotaCommitTask;volume:creat
e' (2a8b14e5-64a8-42b2-88a0-519158f2adc7) transitioned into state 'SUCCESS' from sta
te 'RUNNING' with result: {'volume_properties': {'status': 'creating', 'volume_type_
id': 'u7cd531f0-4fa2-4a2c-9582-4564cb4f54c3', 'multiattach': False, 'reservations':
['61b28b9d-7265-43dd-8530-3b4dde935d3e', '3282f834-2083-4d5b-93ec-530a398d1161', '2c
fd6df0-729f-4b60-a795-248eeae9c24e', 'b314b246-1b72-425b-a645-d47d3794abba'], 'sourc
e_valid': None, 'volume_metadata': [], 'qos_specs': None, 'consistencygroup_id': Non
e, 'replication_status': 'disabled', 'snapshot_id': None, 'display_name': 'u'vol-1',
'id': '1e7f6bd7-cell1-4a73-b95e-aabd65a5b188', 'size': 1, 'user_id': 'u'b02f995f9b7343
84a5662c8d30d661e4', 'source_replica_id': None, 'availability_zone': 'u'nova', 'attach
_status': 'detached', 'display_description': 'u', 'cgsnapshot_id': None, 'encryption
_key_id': None, 'volume_admin_metadata': [], 'project_id': 'u'aas1b851d2a54484b6f3984
ab2c5d4e4', 'metadata': {}}, 'task_receiver /usr/local/lib/python2.7/dist-packages/
taskflow/listeners/logging.py:178

```

图 8-37

最后 VolumeCastTask 是向 cinder-scheduler 发送消息, 开始调度工作, 如图 8-38 所示。

```

2016-01-27 17:07:23.856 DEBUG cinder.volume.api [req-a2561b49-a7ca-45a4-8f4a-09118c8
13be1 None] Task 'cinder.volume.flows.api.create_volume.VolumeCastTask;volume:create
' (5e4740ce-7b8b-477a-b6bd-11490c4a43dc) transitioned into state 'RUNNING' from sta
te 'PENDING' task_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listener
s/logging.py:189
2016-01-27 17:07:23.861 DEBUG cinder.volume.api [req-a2561b49-a7ca-45a4-8f4a-09118c8
13be1 None] Task 'cinder.volume.flows.api.create_volume.VolumeCastTask;volume:create
' (5e4740ce-7b8b-477a-b6bd-11490c4a43dc) transitioned into state 'SUCCESS' from sta
te 'RUNNING' with result: None task_receiver /usr/local/lib/python2.7/dist-packages
taskflow/listeners/logging.py:178

```

图 8-38

至此, Flow volume\_create\_api 已经完成, 状态由 RUNNING 变为 SUCCESS, volume 创建成功, 日志如图图 8-39 所示。

```

2016-01-27 17:07:23.863 DEBUG cinder.volume.api [req-a2561b49-a7ca-45a4-8f4a-09118c8
13be1 None] Flow 'volume_create_api' (ddfb7238-a039-465c-b7e5-e40ee83155d9) transiti
oned into state 'SUCCESS' from state 'RUNNING' flow_receiver /usr/local/lib/python2
.7/dist-packages/taskflow/listeners/logging.py:140
2016-01-27 17:07:23.864 INFO cinder.volume.api [req-a2561b49-a7ca-45a4-8f4a-09118c81
3be1 None] [volume-1e7f6bd7-cell1-4a73-b95e-aabd65a5b188] volume created successfully

```

图 8-39

需要特别注意的是, “volume 创建成功”只是指 cinder-api 已经成功处理了 volume create

请求，将消息发给了 cinder-scheduler，但并不意味 volume 在存储节点上已经成功创建，这一点是容易引起误解的。我们可以通过 cinder-volume 创建 volume 日志的时间戳验证。

### (2) cinder-api 发送消息

cinder-api 向 RabbitMQ 发送了一条消息：“让 cinder-scheduler 创建一个 volume”。

前面我们提到消息是由 VolumeCastTask 发出的，因为 VolumeCastTask 没有打印相关日志，我们只能通过源代码/opt/stack/cinder/cinder/volume/flows/api/create\_volume.py 查看，方法为 create\_volume，如图 8-40 所示。

```

19 self.scheduler_rpcapi.create_volume(
20     context,
21     CONF.volume_topic,
22     volume_id,
23     snapshot_id=snapshot_id,
24     image_id=image_id,
25     request_spec=request_spec,
26     filter_properties=filter_properties)

```

图 8-40

### (3) cinder-scheduler 执行调度

cinder-scheduler 执行调度算法，通过 Filter 和 Weigher 挑选最优的存储节点，日志为 /opt/stack/logs/c-sch.log。cinder-scheduler 通过 Flow volume\_create\_scheduler 执行调度工作，如图 8-41 所示。

```

2016-01-27 17:07:23.871 DEBUG cinder.scheduler.manager [req-a188d1b1-4fe3-41725-8bb6-884a1ed73da2 admin] Flow volume_create_scheduler (37a38421-6b62-4b42-8cd5-fd01c9471043) transitioned into state RUNNING from state PENDING flow_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:140

```

图 8-41

该 Flow 依次执行 ExtractSchedulerSpecTask 和 ScheduleCreateVolumeTask，如图 8-42 所示。

```

2016-01-27 17:07:23.874 DEBUG cinder.scheduler.manager [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Task cinder.scheduler.flows.create_volume.ExtractSchedulerSpecTask;volume:create (016e9e64-2f06-4e6a-ae66-589434da2f0e) transitioned into state RUNNING from state PENDING task_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:189
2016-01-27 17:07:23.876 DEBUG cinder.scheduler.manager [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Task cinder.scheduler.flows.create_volume.ExtractSchedulerSpecTask;volume:create (016e9e64-2f06-4e6a-ae66-589434da2f0e) transitioned into state SUCCESS from state RUNNING with result {'request_spec': {'source_replica_id': None, u'volume_properties': {'u'volume_metadata': [], u'availability_zone': u'nova', u'reservations': [u'61b28b9d-7265-43dd-8530-3b4dde933d3e', u'3282f834-2083-4d5b-93ec-530a398d1161', u'2cfdd6df0-729f-4b60-a795-248eeae9c24e', u'b314b246-1b72-425b-a645-d47d3794abba'], u'replication_status': u'disabled', u'sn

```

图 8-42

主要的 filter 和 weighting 工作由 ScheduleCreateVolumeTask 完成，如图 8-43 所示。

```

2016-01-27 17:07:23.878 DEBUG cinder.scheduler.manager [req-a188d1b1-4fe3-4b25-8
bb6-884a1ed73da2 admin] Task 'cinder.scheduler.flows.create_volume.Scheduledcreat
evolumeTask;volume:create' (6184cc1b-8935-4ff6-88a3-012232e676f1) transitioned i
nto state 'RUNNING' from state 'PENDING' task_receiver /usr/local/lib/python2.7
/dist-packages/taskflow/listeners/logging.py:189
2016-01-27 17:07:23.885 DEBUG cinder.openstack.common.scheduler.base_filter [req
-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Starting with 1 host(s) get_filtere
d_objects /opt/stack/cinder/cinder/openstack/common/scheduler/base_filter.py:77
2016-01-27 17:07:23.886 DEBUG cinder.openstack.common.scheduler.base_filter [req
-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Filter AvailabilityZoneFilter retur
ned 1 host(s) get_filtered_objects /opt/stack/cinder/cinder/openstack/common/sch
eduler/base_filter.py:94
2016-01-27 17:07:23.886 DEBUG cinder.scheduler.filters.capacity_filter [req-a188
d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Space information for volume creation on
1 host devstack-controller@lvmdriver-1#lvmdriver-1 (requested / avail): 1/30.0 ho
st_passes /opt/stack/cinder/cinder/scheduler/filters/capacity_filter.py:122
2016-01-27 17:07:23.887 DEBUG cinder.openstack.common.scheduler.base_filter [req
-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Filter CapacityFilter returned 1 ho
st(s) get_filtered_objects /opt/stack/cinder/cinder/openstack/common/scheduler/b
ase_filter.py:94
2016-01-27 17:07:23.887 DEBUG cinder.openstack.common.scheduler.base_filter [req
-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Filter CapabilitiesFilter returned
1 host(s) get_filtered_objects /opt/stack/cinder/cinder/openstack/common/schedul
er/base_filter.py:94
2016-01-27 17:07:23.887 DEBUG cinder.scheduler.filter_scheduler [req-a188d1b1-4f
e3-4b25-8bb6-884a1ed73da2 admin] Filtered [host 'devstack-controller@lvmdriver-1
#lvmdriver-1': free_capacity_gb: 30.0, pools: None] get_weighted_candidates /op
t/stack/cinder/cinder/scheduler/filter_scheduler.py:310
2016-01-27 17:07:23.888 DEBUG cinder.scheduler.filter_scheduler [req-a188d1b1-4f
e3-4b25-8bb6-884a1ed73da2 admin] Choosing devstack-controller@lvmdriver-1#lvmdri
ver-1 choose_top_host /opt/stack/cinder/cinder/scheduler/filter_scheduler.py:42
9
2016-01-27 17:07:23.945 DEBUG cinder.scheduler.manager [req-a188d1b1-4fe3-4b25-8
bb6-884a1ed73da2 admin] Task 'cinder.scheduler.flows.create_volume.Scheduledcreat
evolumeTask;volume:create' (6184cc1b-8935-4ff6-88a3-012232e676f1) transitioned i
nto state 'SUCCESS' from state 'RUNNING' with result None task_receiver /usr/
local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:178

```

图 8-43

经过 AvailabilityZoneFilter、CapacityFilter、CapabilitiesFilter 和 CapacityWeigher 的层层筛选, 最终选择了存储节点 devstack-controller@lvmdriver-1#lvmdriver-1。Flow volume\_create\_scheduler 完成调度, 状态变为 SUCCESS, 如图 8-44 所示。

```

2016-01-27 17:07:23.946 DEBUG cinder.scheduler.manager [req-a188d1b1-4fe3-4b25-8
bb6-884a1ed73da2 admin] Flow 'volume_create_scheduler' (37a38421-6b62-4b42-8cd5-
fd01c9471043) transitioned into state 'SUCCESS' from state 'RUNNING' flow_recei
ver /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:140

```

图 8-44

#### (4) cinder-scheduler 发送消息

cinder-scheduler 发送消息给 cinder-volume, 让其创建 volume。源码查看 /opt/stack/cinder/cinder/scheduler/filters\_scheduler.py, 方法为 schedule\_create\_volume, 如图 8-45 所示。

```

82
83 def schedule_create_volume(self, context, request_spec, filter_properties):
84     weighed_host = self._schedule(context, request_spec,
85                                   filter_properties)
86
87     if not weighed_host:
88         raise exception.InvalidHost(reason=...)
89
90     host = weighed_host.obj.host
91     volume_id = request_spec[...]
92
93     updated_volume = driver.volume_update_db(context, volume_id, host)
94     self._post_select_populate_filter_properties(filter_properties,
95                                                  weighed_host.obj)
96
97     filter_properties.pop(...)
98
99     self.volume_rpcapi.create_volume(context, updated_volume, host,
100                                     request_spec, filter_properties,
101                                     allow_reschedule=True)
102

```

图 8-45

#### (5) cinder-volume 执行操作

cinder-volume 通过 driver 创建 volume, 日志为 /opt/stack/logs/c-vol.log。与 cinder-api 和



cinder-scheduler 执行方式类似, cinder-volume 也启动了一个 Flow 来完成 volume 创建工作, Flow 的名称为 volume\_create\_manager, 如图 8-46 所示。

```
2016-01-27 17:07:23.971 DEBUG cinder.volume.manager [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Flow 'volume_create_manager' (67031072-ad1d-4b14-ae19-096f3c5b092c) transitioned into state 'RUNNING' from state 'PENDING' task_receiver: /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:140
```

图 8-46

volume\_create\_manager 首先执行 ExtractVolumeRefTask、OnFailureRescheduleTask、ExtractVolumeSpecTask、NotifyVolumeActionTask 为 volume 创建做准备, 如图 8-47 所示。

```
2016-01-27 17:07:23.977 DEBUG cinder.volume.manager [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Task 'cinder.volume.flows.manager.create_volume.ExtractVolumeRefTask;volume:create' (1941d770-5b47-4850-a070-447b3d1bf4b) transitioned into state 'RUNNING' from state 'PENDING' task_receiver: /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:189
2016-01-27 17:07:24.033 DEBUG cinder.volume.manager [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Task 'cinder.volume.flows.manager.create_volume.ExtractVolumeRefTask;volume:create' (1941d770-5b47-4850-a070-447b3d1bf4b) transitioned into state 'SUCCESS' from state 'RUNNING' with result: <cinder.db.sqlalchemy.models.Volume object at 0x7f713b5bf590> task_receiver: /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:178
```

```
2016-01-27 17:07:24.035 DEBUG cinder.volume.manager [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Task 'cinder.volume.flows.manager.create_volume.OnFailureRescheduleTask;volume:create' (803103fa-d50c-40c4-9b85-885791208d49) transitioned into state 'RUNNING' from state 'PENDING' task_receiver: /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:189
2016-01-27 17:07:24.036 DEBUG cinder.volume.manager [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Task 'cinder.volume.flows.manager.create_volume.OnFailureRescheduleTask;volume:create' (803103fa-d50c-40c4-9b85-885791208d49) transitioned into state 'SUCCESS' from state 'RUNNING' with result: None task_receiver: /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:178
```

```
2016-01-27 17:07:24.038 DEBUG cinder.volume.manager [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Task 'cinder.volume.flows.manager.create_volume.ExtractVolumeSpecTask;volume:create' (56e5879b-7726-4bce-b66a-7704d13d812e) transitioned into state 'RUNNING' from state 'PENDING' task_receiver: /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:189
2016-01-27 17:07:24.040 DEBUG cinder.volume.manager [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Task 'cinder.volume.flows.manager.create_volume.ExtractVolumeSpecTask;volume:create' (56e5879b-7726-4bce-b66a-7704d13d812e) transitioned into state 'SUCCESS' from state 'RUNNING' with result: {'status': 'creating', 'volume_size': 1, 'volume_name': 'volume-1e7f6bd7-cell1-4a73-b95e-aab65a5b188', 'type': 'raw', 'volume_id': '1e7f6bd7-cell1-4a73-b95e-aab65a5b188'} task_receiver: /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:178
```

```
2016-01-27 17:07:24.041 DEBUG cinder.volume.manager [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Task 'cinder.volume.flows.manager.create_volume.NotifyVolumeActionTask;volume:create, create.start' (3cb9bc14-ae99-4352-9f5c-a04f72c82dd6) transitioned into state 'RUNNING' from state 'PENDING' task_receiver: /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:189
2016-01-27 17:07:24.139 DEBUG cinder.volume.manager [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Task 'cinder.volume.flows.manager.create_volume.NotifyVolumeActionTask;volume:create, create.start' (3cb9bc14-ae99-4352-9f5c-a04f72c82dd6) transitioned into state 'SUCCESS' from state 'RUNNING' with result: None task_receiver: /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:178
```

图 8-47



接下来 CreateVolumeFromSpecTask 执行 volume 创建任务,如图 8-48 所示。

```
2016-01-27 17:07:24.140 DEBUG cinder.volume.manager [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Task 'cinder.volume.flows.manager.create_volume.createVolumeFromSpecTask;volume:create' (565a4aa1-1af7-4473-8673-2626e808ba63) transitioned into state 'RUNNING' from state 'PENDING' task_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:189
2016-01-27 17:07:24.141 INFO cinder.volume.flows.manager.create_volume [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] volume 1e7f6bd7-ce11-4a73-b95e-aabd65a5b188: being created as raw with specification: {'status': 'u'creating', 'volume_size': 1, 'volume_name': 'u'volume-1e7f6bd7-ce11-4a73-b95e-aabd65a5b188'}
2016-01-27 17:07:24.141 DEBUG oslo_concurrency.processutils [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Running cmd (subprocess): lvcreate -n volume-1e7f6bd7-ce11-4a73-b95e-aabd65a5b188 stack-volumes-lvmdriver-1 -L 1g execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-01-27 17:07:24.217 DEBUG oslo_concurrency.processutils [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] CMD 'lvcreate -n volume-1e7f6bd7-ce11-4a73-b95e-aabd65a5b188 stack-volumes-lvmdriver-1 -L 1g' returned: 0 in 0.075s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
2016-01-27 17:07:24.221 DEBUG cinder.volume.manager [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Task 'cinder.volume.flows.manager.create_volume.createVolumeFromSpecTask;volume:create' (565a4aa1-1af7-4473-8673-2626e808ba63) transitioned into state 'SUCCESS' from state 'RUNNING' with result '<cinder.db.sqlalchemy.models.Volume object at 0x7f713b5bf590>' task_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:178
```

图 8-48

因为 volume provider 为 LVM, CreateVolumeFromSpecTask 通过 lvcreate 命令在 VG stack-volumes-lvmdriver-1 中创建了一个 1G 的 LV, cinder-volume 将这个 LV 作为 volume。

新创建的 LV 命名为“volume-1e7f6bd7-ce11-4a73-b95e-aabd65a5b188”, 其格式为“volume-<volume\_id>”, 如图 8-49 所示。

```
root@devstack-controller:~# cinder list
+-----+-----+
| ID                                           | Status |
+-----+-----+
| 1e7f6bd7-ce11-4a73-b95e-aabd65a5b188      | available |
+-----+-----+

root@devstack-controller:~# lvsdisplay
--- Logical volume ---
LV Path                /dev/stack-volumes-lvmdriver-1/volume-1e7f6bd7-ce11-4a73-b95e-aabd65a5b188
LV Name                 volume-1e7f6bd7-ce11-4a73-b95e-aabd65a5b188
VG Name                 stack-volumes-lvmdriver-1
LV UUID                 EH799U-MC5o-5Dds-8lyU-o1M5-TM6F-GGfBny
LV write Access         read/write
LV Creation host, time  devstack-controller, 2016-01-27 17:07:24 +0800
LV Status                available
# open                   0
LV Size                 1.00 GiB
Current LE               256
Segments                 1
Allocation                inherit
Read ahead sectors      auto
- currently set to      256
Block device             252:2
```

图 8-49

最后, CreateVolumeOnFinishTask 完成扫尾工作,如图 8-50 所示。

```
2016-01-27 17:07:24.224 DEBUG cinder.volume.manager [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] task 'cinder.volume.flows.manager.create_volume.CreateVolumeOnFinishTask;volume:create,create,end' (23517aab-f16d-4a40-92e5-b59ad7b5a108) transitioned into state 'RUNNING' from state 'PENDING' _task_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:189
2016-01-27 17:07:24.606 INFO cinder.volume.flows.manager.create_volume [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] volume volume-1e7f6bd7-cell-4a73-b95e-aabd65a5b188 (1e7f6bd7-cell-4a73-b95e-aabd65a5b188): created successfully
2016-01-27 17:07:24.608 DEBUG cinder.volume.manager [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] task 'cinder.volume.flows.manager.create_volume.CreateVolumeOnFinishTask;volume:create,create,end' (23517aab-f16d-4a40-92e5-b59ad7b5a108) transitioned into state 'SUCCESS' from state 'RUNNING' with result 'None' _task_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:178
```

图 8-50

至此，volume 成功创建，Flow volume\_create\_manager 结束，如图 8-51 和图 8-52 所示。

```
2016-01-27 17:07:24.610 DEBUG cinder.volume.manager [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] Flow 'volume_create_manager' (67031072-add1-4b14-ae19-096f3c5b092c) transitioned into state 'SUCCESS' from state 'RUNNING' _flow_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:140
2016-01-27 17:07:24.610 INFO cinder.volume.manager [req-a188d1b1-4fe3-4b25-8bb6-884a1ed73da2 admin] [volume-1e7f6bd7-cell-4a73-b95e-aabd65a5b188] created volume successfully.
```

图 8-51

Volumes

Volumes

Volume Snapshots

<input type="checkbox"/>	Name	Description	Size	Status	Type
<input type="checkbox"/>	vol-1	-	1GB	Available	lvmdriver-1

Displaying 1 item

图 8-52

3. Attach

Volume 的最主要用途是作为虚拟硬盘提供给 instance 使用。Volume 是通过 Attach 操作挂载到 instance 上的。本节我们就来详细讨论 Cinder 是如何实现 Attach 的。

上一节我们成功创建了基于 LVM provider 的 volume。每个 volume 实际上是存储节点上 VG 中的一个 LV。

那么问题来了：存储节点上本地的 LV 如何挂载到计算节点的 instance 上呢？通常情况存储节点和计算节点是不同的物理节点。

解决方案是使用 iSCSI，如图 8-53 所示。

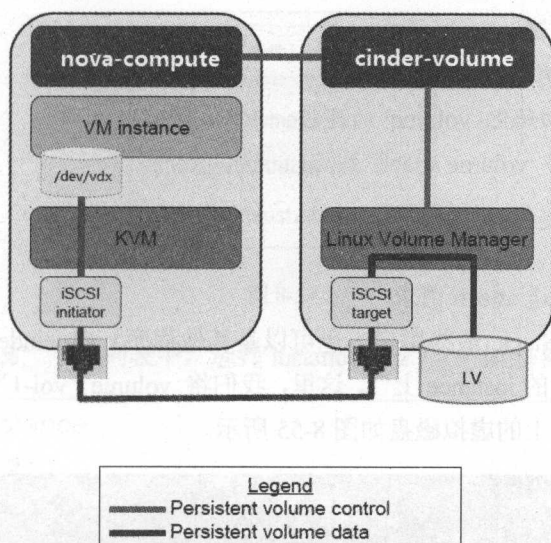


图 8-53

iSCSI 是 Client-Server 架构，有 target 和 initiator 两个术语。

- Target

提供 iSCSI 存储资源的设备，简单地说，就是 iSCSI 服务器。

- Initiator

使用 iSCSI 存储资源的设备，也就是 iSCSI 客户端。

Initiator 需要与 target 建立 iSCSI 连接，执行 login 操作，然后就可以使用 target 上面的块存储设备了。

Target 提供的块存储设备支持多种实现方式，我们实验环境中使用的是 LV。

Cinder 的存储节点 cinder-volume 默认使用 tgt 软件来管理和监控 iSCSI target，在计算节点 nova-compute 使用 iscsiadm 执行 initiator 相关操作。

下面来看看 Attach 操作的流程图，如图 8-54 所示。

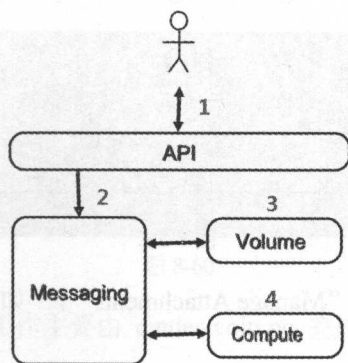


图 8-54

- 向 cinder-api 发送 attach 请求。
- cinder-api 发送消息。
- cinder-volume 初始化 volume 的连接。
- nova-compute 将 volume attach 到 instance。

下面我们详细讨论每一个步骤。

### (1) 向 cinder-api 发送 attach 请求

客户（可以是 OpenStack 最终用户，也可以是其他程序）向 cinder-api 发送请求：“请将这个 volume attach 到指定的 instance 上”。这里，我们将 volume “vol-1” attach 到 instance “c2” 上。attach 操作之前，c2 上的虚拟磁盘如图 8-55 所示。

```
login as 'cirros' user: default password: 'cubswin:)' use 'sudo' for root
c2 login: cirros
Password:
$ sudo fdisk -l

Disk /dev/vda: 41 MB, 41126400 bytes
255 heads, 63 sectors/track, 5 cylinders, total 80325 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

   Device Boot      Start         End      Blocks   Id  System
   /dev/vda1        16065        80324        32130    83  Linux

Disk /dev/vdb: 67 MB, 67168864 bytes
16 heads, 63 sectors/track, 130 cylinders, total 131072 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

   Device Boot      Start         End      Blocks   Id  System
```

图 8-55

进入 GUI 操作菜单 Project → Compute → Volumes，如图 8-56 所示。

Volumes						
Volume Snapshots						
<input type="checkbox"/>	Name	Description	Size	Status	Type	Attached To
<input type="checkbox"/>	vol-1	-	1GB	Available	lvmdriver-1	

图 8-56

选择 volume “vol-1”，单击“Manage Attachments”，如图 8-57 所示。



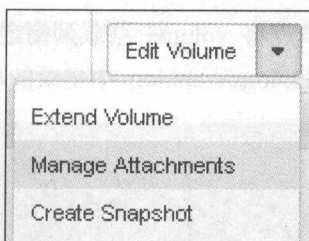


图 8-57

在“Attach to Instance”下拉列表中，选择 instance “c2”，如图 8-58 所示。

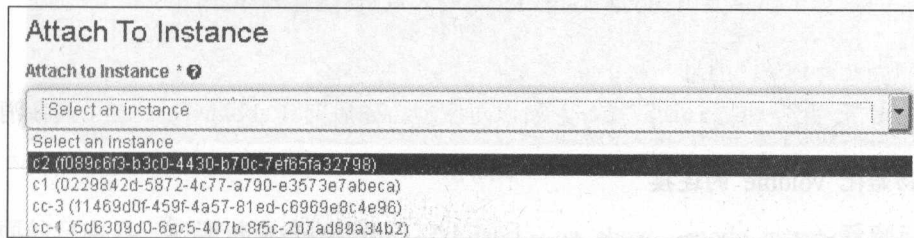


图 8-58

单击“Attach Volume”，如图 8-59 所示。

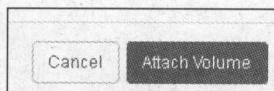


图 8-59

cinder-api 将接收到 attach volume 的请求，attach 请求实际上包含两个步骤：

- 初始化 volume 的连接

Volume 创建后，只是在 volume provider 中创建了相应存储对象（比如 LV），这时计算节点是无法使用的。Cinder-volume 需要以某种方式将 volume export 出来，计算节点才能够访问得到。这个 export 的过程就是“初始化 volume 的连接”。

下面是 cinder-api 的日志文件 /opt/stack/logs/c-api.log 中记录的相关信息，如图 8-60 所示。

```
2016-02-05 22:37:18.857 INFO cinder.api.openstack.wsgi [req-a013668f-e5b9-4f64-b89f-ed216c925052 admin] POST http://192.168.104.10:8776/v2/aa81b851d2a54484b6f3984ab2c5d4e4/volumes/1e7f6bd7-ce11-4a73-b95e-aabd65a5b188/attachment
2016-02-05 22:37:18.857 DEBUG cinder.api.openstack.wsgi [req-a013668f-e5b9-4f64-b89f-ed216c925052 admin] Action body: {'os_initialize_connection': {'connector': {'initiator': 'iqn.1993-08.org.debian:01:ee3af514d83', 'ip': '192.168.104.11', 'platform': 'x86_64', 'host': 'devstack-computel', 'os_type': 'linux2', 'multipath': false}}} get_method /opt/stack/cinder/cinder/api/openstack/wsgi.py:1088
```

图 8-60

Initialize\_connection 的具体工作主要由 cinder-volume 完成，将在后面详细讨论。

- Attach volume

初始化 volume 连接后, 计算节点将 volume 挂载到指定的 instance, 完成 attach 操作。下面是 cinder-api 的日志文件 /opt/stack/logs/c-api.log 中记录的相关信息, 如图 8-61 所示。

```
2016-02-05 22:37:22.785 INFO cinder.api.openstack.wsgi [req-16b5e188-e228-4dee-8370-259730c6e667 admin] POST http://192.168.104.10:8776/v2/aa81b851d2a54484b6f3984ab2c5d4e4/volumes/1e7f6bd7-4ce11-4a73-b95e-aabd65a5b188/attachment
2016-02-05 22:37:22.786 DEBUG cinder.api.openstack.wsgi [req-16b5e188-e228-4dee-8370-259730c6e667 admin] Action body: {"os-attach": {"instance_uuid": "f089c6f3-b3c0-4430-b70c-7ef65fa32798", "mountpoint": "/dev/vdb", "mode": "rw"}} get_method: /opt/stack/cinder/cinder/api/openstack/wsgi.py:1088
```

图 8-61

Attach 的具体工作主要由 nova-compute 完成, 也将在后面详细讨论。

## (2) cinder-api 发送消息

cinder-api 分两步完成 attach 操作, 所以对应地会先后向 RabbitMQ 发送了两条消息:

### ● 初始化 volume 的连接

cinder-api 没有打印发送消息的日志, 只能通过源代码查看, 源文件为 /opt/stack/cinder/cinder/volume/api.py, 方法为 initialize\_connection, 如图 8-62 所示。

```
655 def initialize_connection(self, context, volume, connector):
656     if volume['status'] == 'available':
657         LOG.info('Initializing connection for volume %s', volume['id'])
658         msg = {'volume_id': volume['id'], 'resource': volume}
659         raise exception.InvalidVolume(reason=msg)
660     init_results = self.volume_rpcapi.initialize_connection(context, volume, connector)
661     return init_results
```

图 8-62

### ● Attach volume

cinder-api 没有打印发送消息的日志, 只能通过源代码查看, 源文件为 /opt/stack/cinder/cinder/volume/api.py, 方法为 attach, 如图 8-63 所示。

```
612 def attach_volume(self, context, volume, instance_uuid, host_name,
613                  mountpoint, mode):
614     if volume['status'] != 'available':
615         LOG.info('Volume %s is not available for attachment', volume['id'])
616         msg = {'volume_id': volume['id'], 'resource': volume}
617         raise exception.InvalidVolume(reason=msg)
618     volume_metadata = self.get_volume_admin_metadata(context.elevated(), volume)
619     if volume_metadata is None:
620         raise exception.InvalidVolume(reason='Volume %s has no metadata' % volume['id'])
621     NOTE
622     self.update_volume_admin_metadata(context.elevated(), volume, volume_metadata)
623     volume_metadata['attach_info'] = {'instance_uuid': instance_uuid, 'host_name': host_name, 'mountpoint': mountpoint, 'mode': mode}
624     if volume_metadata['attach_info'] == {} and mode != '':
625         raise exception.InvalidVolumeAttachMode(mode=mode, volume_id=volume['id'])
626     attach_results = self.volume_rpcapi.attach_volume(context, volume, instance_uuid, host_name, mountpoint, mode)
```

图 8-63

### (3) cinder-volume 初始化 volume 的连接

cinder-volume 接收到 initialize\_connection 消息后,会通过 tgt 创建 target,并将 volume 所对应的 LV 通过 target export 出来。日志为 /opt/stack/logs/c-vol.log,如图 8-64 所示。

```
2016-02-05 22:37:19.022 DEBUG cinder.volume.targets.tgt [req-a013668f-e5b9-4f64-b89f-ed216c925052 admin] Creating iscsi_target for volume ID: volume-1e7f6bd7-ce11-4a73-b95e-aabd65a5b188 create_iscsi_target /opt/stack/cinder/cinder/volume/targets/tgt.py:190
2016-02-05 22:37:19.022 DEBUG cinder.volume.targets.tgt [req-a013668f-e5b9-4f64-b89f-ed216c925052 admin] Created volume path /opt/stack/data/cinder/volumes/volume-1e7f6bd7-ce11-4a73-b95e-aabd65a5b188,
content:
<target iqn.2010-10.org.openstack:volume-1e7f6bd7-ce11-4a73-b95e-aabd65a5b188>
    backing-store /dev/stack-volumes-lvmdriver-1/volume-1e7f6bd7-ce11-4a73-b95e-aabd65a5b188
    driver iscsi
    incominguser EbrdxehuxVLGJ29bQbXB QttyZjUr75wup19N
    write-cache on
</target>
create_iscsi_target /opt/stack/cinder/cinder/volume/targets/tgt.py:202
```

图 8-64

下面的日志显示:通过命令 `tgtadm --lld iscsi --op show --mode target` 看到已经将 1GB (1074MB) 的 LV /dev/stack-volumes-lvmdriver-1/volume-1e7f6bd7-ce11-4a73-b95e-aabd65a5b188 通过 Target 1 export 出来了,如图 8-65 所示。

```
2016-02-05 22:37:19.271 DEBUG oslo_concurrency.processutils [req-a013668f-e5b9-4f64-b89f-ed216c925052 admin] CMD "tgtadm --lld iscsi --op show --mode target" returned: 0 in 0.012s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
2016-02-05 22:37:19.273 DEBUG cinder.volume.targets.tgt [req-a013668f-e5b9-4f64-b89f-ed216c925052 admin] Targets after update:
Target 1: iqn.2010-10.org.openstack:volume-1e7f6bd7-ce11-4a73-b95e-aabd65a5b188
System information:
  Driver: iscsi
  State: ready
I_T nexus information:
LUN information:
  LUN: 0
    Type: controller
    SCSI ID: IET 00010000
    SCSI SN: beaf10
    Size: 0 MB, Block size: 1
    Online: Yes
    Removable media: No
    Prevent removal: No
    Readonly: No
    SWP: No
    Thin-provisioning: No
    Backing store type: null
    Backing store path: None
    Backing store flags:
  LUN: 1
    Type: disk
    SCSI ID: IET 00010001
    SCSI SN: beaf11
    Size: 1074 MB, Block size: 512
    Online: Yes
    Removable media: No
    Prevent removal: No
    Readonly: No
    SWP: No
    Thin-provisioning: No
    Backing store type: rdwr
    Backing store path: /dev/stack-volumes-lvmdriver-1/volume-1e7f6bd7-ce11-4a73-b95e-aabd65a5b188
    Backing store flags:
```

图 8-65



Initialize connection 完成, 如图 8-66 所示。

```
2016-02-05 22:37:19.603 INFO cinder.volume.manager [req-1ba3173c-f803-46
16-b6c7-669eb8a6dfde None] [volume-1e7f6bd7-cell1-4a73-b95e-aabd65a5b188]
Initialize volume connection completed successfully.
```

图 8-66

(4) nova-compute 将 volume attach 到 instance

计算节点作为 iSCSI initiator 访问存储节点 Iscsi Target 上的 volume, 并将其 attach 到 instance。日志文件为 /opt/stack/logs/n-cpu.log, 如图 8-67 所示。

```
2016-02-05 22:37:20.888 DEBUG nova.virt.libvirt.volume.iscsi [req-43a94
801-38f4-4d50-9670-dba55a6a8118 admin admin] Calling os-brick to attach
iscsi volume connect_volume /opt/stack/nova/nova/virt/libvirt/volume/t
scsi.py:83
```

图 8-67

nova-compute 依次执行 iscsiadm 的 new、update、login、rescan 操作访问 target 上的 volume, 如图 8-68 所示。

```
2016-02-05 22:37:21.037 DEBUG oslo_concurrency.processutils [req-43a948
01-38f4-4d50-9670-dba55a6a8118 admin admin] CMD "iscsiadm -m node -T iq
n.2010-10.org.openstack:volume-1e7f6bd7-cell1-4a73-b95e-aabd65a5b188 -p
192.168.104.10:3260 --interface=default --op new" returned: 0 in 0.102s
execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/proces
sutils.py:280
```

```
2016-02-05 22:37:21.077 DEBUG oslo_concurrency.processutils [req-43a948
01-38f4-4d50-9670-dba55a6a8118 admin admin] CMD "iscsiadm -m node -T iq
n.2010-10.org.openstack:volume-1e7f6bd7-cell1-4a73-b95e-aabd65a5b188 -p
192.168.104.10:3260 --op update -n node.session.auth.password -v ***" r
eturned: 0 in 0.013s execute /usr/local/lib/python2.7/dist-packages/osl
o_concurrency/processutils.py:280
```

```
2016-02-05 22:37:22.596 DEBUG oslo_concurrency.processutils [req-43a948
01-38f4-4d50-9670-dba55a6a8118 admin admin] CMD "iscsiadm -m node -T iq
n.2010-10.org.openstack:volume-1e7f6bd7-cell1-4a73-b95e-aabd65a5b188 -p
192.168.104.10:3260 --login" returned: 0 in 1.506s execute /usr/local/l
ib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
```

```
2016-02-05 22:37:22.666 DEBUG oslo_concurrency.processutils [req-43a948
01-38f4-4d50-9670-dba55a6a8118 admin admin] CMD "iscsiadm -m node -T iq
n.2010-10.org.openstack:volume-1e7f6bd7-cell1-4a73-b95e-aabd65a5b188 -p
192.168.104.10:3260 --rescan" returned: 0 in 0.048s execute /usr/local/
lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
```

图 8-68

计算节点将 iSCSI target 上的 volume 识别为一个磁盘文件, 如图 8-69 所示。

```
root@devstack-compute1:~# ls -l /dev/disk/by-path/ip-192.168.104.10:326
0-iscsi-iqn.2010-10.org.openstack:volume-1e7f6bd7-cell1-4a73-b95e-aabd65
a5b188-lun-1
lrwxrwxrwx 1 root root 9 Feb 5 23:36 /dev/disk/by-path/ip-192.168.104.
10:3260-iscsi-iqn.2010-10.org.openstack:volume-1e7f6bd7-cell1-4a73-b95e-
aabd65a5b188-lun-1 -> ../../sdb
```

图 8-69

然后通过更新 instance 的 XML 配置文件将 volume 映射给 instance, 如图 8-70 所示。



```

2016-02-05 22:37:23.668 DEBUG nova.virt.libvirt.volume.iscsi [req-43a94
801-38f4-4d50-9670-dba55a6a8118 admin admin] Attached iscsi volume {'pa
th': u'/dev/disk/by-path/ip-192.168.104.10:3260-iscsi-qn.2010-10.org.o
penstack:volume-1e7f6bd7-cell-4a73-b95e-aabd65a5b188-lun-1', 'type': 'b
lock'}, connect_volume /opt/stack/nova/nova/virt/libvirt/volume/iscsi.py
:85
2016-02-05 22:37:23.674 DEBUG nova.virt.libvirt.config [req-43a94801-38
f4-4d50-9670-dba55a6a8118 admin admin] Generated XML ('<disk type="bloc
k" device="disk">\n  <driver name="qemu" type="raw" cache="none"/>\n  <
source dev="/dev/disk/by-path/ip-192.168.104.10:3260-iscsi-qn.2010-10.
org.openstack:volume-1e7f6bd7-cell-4a73-b95e-aabd65a5b188-lun-1"/>\n  <
target bus="virtio" dev="vdb"/>\n  <serial>1e7f6bd7-cell-4a73-b95e-aabd
65a5b188</serial>\n</disk>\n') to_xml /opt/stack/nova/nova/virt/libvi
rt/config.py:82

```

图 8-70

我们也可以通过 `virsh edit <instance>` 查看更新后的 XML，如图 8-71 所示。

```

<disk type="disk" device="disk">
  <driver name="qemu" type="raw" cache="none" />
  <source dev="/dev/disk/by-path/ip-192.168.104.10:3260-iscsi-qn.2010-10.org.openstack:volume-1e7f6bd7-cell-4a73-b95e-aabd65a5b188-lun-1" />
  <target dev="vdb" bus="virtio" />
  <serial>1e7f6bd7-cell-4a73-b95e-aabd65a5b188</serial>
  <address type="pci" domain="0x0000" bus="0x00" slot="0x00" function="0x00" />
</disk>

```

图 8-71

可以看到，instance 增加了一个类型为 block 的虚拟磁盘，source 就是要 attach 的 volume，该虚拟磁盘的设备名为 vdb。

手工 Shut off 并 Start instance，通过 `fdisk -l` 查看到 volume 已经 attach 上来，设备为 vdb，如图 8-72 所示。

```

Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

   Device Boot      Start         End      Blocks   Id  System
/dev/vda1 *         16065        80324        32130    83  Linux

Disk /dev/vdb: 1073 MB, 1073741824 bytes
16 heads, 63 sectors/track, 2080 cylinders, total 2097152 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

Disk /dev/vdb doesn't contain a valid partition table

Disk /dev/vdc: 67 MB, 67108864 bytes
16 heads, 63 sectors/track, 130 cylinders, total 131072 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

   Device Boot      Start         End      Blocks   Id  System

```

图 8-72

GUI 界面也会更新相关 attach 信息，如图 8-73 所示。

<input type="checkbox"/>	Name	Description	Size	Status	Type	Attached To
<input type="checkbox"/>	vol-1	-	1GB	In-use	lvmdriver-1	Attached to c2 on /dev/vdb

图 8-73

现在如果我们在存储节点执行 `tgt-admin --show --mode target`，会看到计算节点作为 `initiator` 已经连接到 `target 1`。`cinder-volume` 刚刚创建 `target` 的时候是没有 `initiator` 连接的，大家可以将下面的截图与之前的日志做个对比，如图 8-74 所示。

```
root@devstack-controller:~# tgt-admin --show
Target 1: iqn.2010-10.org.openstack:volume-1e7f6bd7-cell1-4a73-b95e-aabd65a5b188
System information:
  Driver: iscsi
  State: ready
I_T nexus information:
  I_T nexus: 1
  Initiator: iqn.1993-08.org.debian:01:eec3af514d83 alias: devstack-computer1
  Connection: 0
  IP Address: 192.168.104.11
LUN information:
  LUN: 0
    Type: controller
    SCSI ID: IET 00010000
    SCSI SN: beaf10
    Size: 0 MB, Block size: 1
    Online: Yes
    Removable media: No
    Prevent removal: No
    Readonly: No
    SWP: No
    Thin-provisioning: No
    Backing store type: null
    Backing store path: None
    Backing store flags:
  LUN: 1
    Type: disk
    SCSI ID: IET 00010001
    SCSI SN: beaf11
    Size: 1074 MB, Block size: 512
    Online: Yes
    Removable media: No
    Prevent removal: No
    Readonly: No
    SWP: No
    Thin-provisioning: No
    Backing store type: rdwr
    Backing store path: /dev/stack-volumes-lvmdriver-1/volume1e7f6bd7-cell1-4a73-b95e-aabd65a5b188
    Backing store flags:
```

图 8-74

以上就是 `attach` 整个操作过程的分析。

4. Detach

与 `Volume Attach` 相对的操作是 `Detach`，就是将 `Volume` 从 `instance` 上卸载下来。如图 8-75 所示是 `Detach` 操作的流程图。

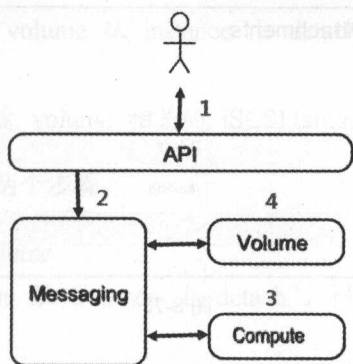


图 8-75

- 向 cinder-api 发送 detach 请求。
- cinder-api 发送消息。
- nova-compute detach volume。
- cinder-volume 删除 target。

下面我们详细讨论每一个步骤。

#### (1) 向 cinder-api 发送 attach 请求

客户(可以是 OpenStack 最终用户,也可以是其他程序)向 cinder-api 发送请求:“请 detach 指定 instance 上的 volume”。这里我们将 detach instance “c2” 上的 volume “vol-1”。

进入 GUI, 操作菜单 Project → Compute → Volumes, 如图 8-76 所示。

<input type="checkbox"/>	Name	Description	Size	Status	Type	Attached To
<input type="checkbox"/>	vol-1	-	1GB	In-use	lvmdriver-1	Attached to c2 on /dev/vdb

图 8-76

选择 volume “vol-1”, 单击 “Manage Attachments”, 如图 8-77 所示。

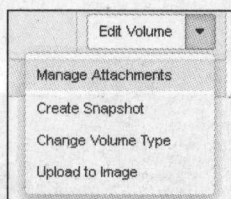


图 8-77

单击 “Detach Volume”, 如图 8-78 所示。

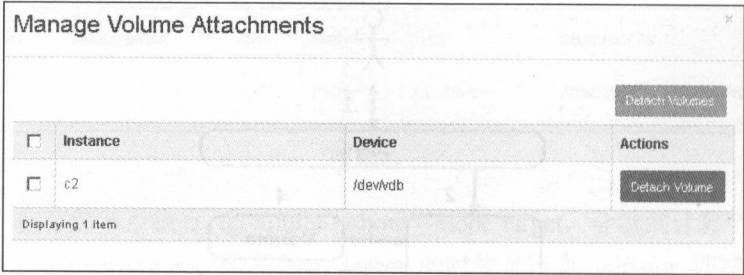


图 8-78

再次确认，如图 8-79 所示。

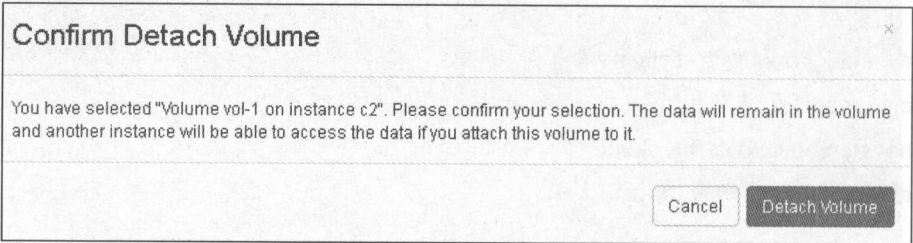


图 8-79

cinder-api 将接收到 detach volume 的请求。日志文件为 /opt/stack/logs/c-api.log，信息如图 8-80 所示。

```
2016-02-07 17:21:01.608 INFO cinder.api.openstack.wsgi [req-546b875f-4d09-41e8-ab13-065a6d6a301a admin] POST http://192.168.104.10:8776/v2/aa81b851d2a54484b6f3984ab2c5d4e4/volumes/1e7f6bd7-ce11-4a73-b95e-aabd65a5b188/action
2016-02-07 17:21:01.608 DEBUG cinder.api.openstack.wsgi [req-546b875f-4d09-41e8-ab13-065a6d6a301a admin] Action body: {"os-detach": {"attachment_id": null}} get_method /opt/stack/cinder/cinder/api/openstack/wsgi.py:1088
```

图 8-80

(2) cinder-api 发送消息

cinder-api 发送消息 detach 消息。

cinder-api 没有打印发送消息的日志，只能通过源代码查看 /opt/stack/cinder/cinder/volume/api.py，方法为 detach。如图 8-81 所示。

```
642 def detach(self, context, volume, attachment_id):
643     if volume:
644         LOG.info(_LI('Detaching volume %s from instance %s'), resource=volume)
645         msg = _('Volume %s is not attached to instance %s') % (volume.id, instance_id)
646         raise exception.InvalidVolume(reason=msg)
647     detach_results = self.volume_rpcapi.detach_volume(context, volume, attachment_id)
648     LOG.info(_LI('Detached volume %s from instance %s'), resource=volume)
649     return detach_results
```

图 8-81

Detach 的操作由 nova-compute 和 cinder-volume 共同完成：



- 首先 nova-compute 将 volume 从 instance 上 detach，然后断开与 iSCSI target 的连接。
- 最后 cinder-volume 删除 volume 相关的 iSCSI target。

后面两个小节将详细讨论这两个步骤。

### (3) nova-compute detach volume

nova-compute 首先将 volume 从 instance 上 detach。日志为 /opt/stack/logs/n-cpu.log，如图 8-82 所示。

```
2016-02-07 17:21:01.526 INFO nova.compute.manager [req-7330c95a-1491-46ca-85a8-21710375fac2 admin admin] [instance: f089c6f3-b3c0-4430-b70c-7ef65fa32798] Detach volume 1e7f6bd7-cel1-4a73-b95e-aabd65a5b188 from mountpoint /dev/vdb
```

图 8-82

这时通过 virsh edit <instance> 可以看到 XML 配置文件中已经不再有 volume 的虚拟磁盘，如图 8-83 所示。

```
<emulator> /usr/bin/qemu-system-x86_64</emulator>
<disk type='file' device='disk'>
  <driver name='qemu' type='raw' cache='on'>
    <source file='disk.img'>
    </source file>
  </driver>
  <target dev='hda' bus='ide'>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x00'>
    </address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x00'>
  </target dev='hda' bus='ide'>
</disk>
<disk type='file' device='disk'>
  <driver name='qemu' type='raw' cache='on'>
    <source file='disk.img'>
    </source file>
  </driver>
  <target dev='hda' bus='ide'>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x00'>
    </address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x00'>
  </target dev='hda' bus='ide'>
</disk>
<controller type='ide' index='0'>
  </controller type='ide' index='0'>
```

图 8-83

接下来断开与 iSCSI target 的连接，如图 8-84 所示。

```
2016-02-07 17:21:01.962 DEBUG nova.virt.libvirt.volume.iscsi [req-7330c95a-1491-46ca-85a8-21710375fac2 admin admin] calling os-brick to detach iscsi
'Volume disconnect volume /opt/stack/nova/nova/virt/libvirt/volume/iscsi.p
v:92
```

图 8-84

具体有下面几个步骤：

将缓存中的数据 Flush 到 volume，如图 8-85 所示。

```
2016-02-07 17:21:01.963 DEBUG os_brick.initiator.linuxscsi [req-7330c95a-1491-46ca-85a8-21710375fac2 admin admin] flushing IO for device /dev/sdb fl
ush_device_io /usr/local/lib/python2.7/dist-packages/os_brick/initiator/li
nuxscsi.py:131
2016-02-07 17:21:01.963 DEBUG oslo_concurrency.processutils [req-7330c95a-1491-46ca-85a8-21710375fac2 admin admin] Running cmd (subprocess): blockde
v --flushbufs /dev/sdb execute /usr/local/lib/python2.7/dist-packages/oslo
_concurrency/processutils.py:250
2016-02-07 17:21:01.975 DEBUG oslo_concurrency.processutils [req-7330c95a-1491-46ca-85a8-21710375fac2 admin admin] cmd 'blockdev --flushbufs /dev/sd
b' returned: 0 in 0.011s execute /usr/local/lib/python2.7/dist-packages/os
lo_concurrency/processutils.py:280
```

图 8-85

删除计算节点上 volume 对应的 SCSI 设备, 如图 8-86 所示。

```
2016-02-07 17:21:01.976 DEBUG os_brick.initiator.linuxscsi [req-7330c95a-1491-46ca-85a8-21710375fac2 admin admin] Remove SCSI device /dev/sdb with /usr/local/lib/python2.7/dist-packages/os_brick/initiator/linuxscsi.py:70
2016-02-07 17:21:01.976 DEBUG oslo_concurrency.processutils [req-7330c95a-1491-46ca-85a8-21710375fac2 admin admin] Running cmd (subprocess): tee -a /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-02-07 17:21:02.014 DEBUG oslo_concurrency.processutils [req-7330c95a-1491-46ca-85a8-21710375fac2 admin admin] CMD "tee -a /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280" returned: 0 in 0.037s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
2016-02-07 17:21:02.015 DEBUG os_brick.initiator.linuxscsi [req-7330c95a-1491-46ca-85a8-21710375fac2 admin admin] Checking to see if SCSI volume /dev/disk/by-path/ip-192.168.104.10:3260-iscsi-qn.2010-10.org.openstack:volume-1e7f6bd7-cell-4a73-b95e-aabd65a5b188-lun-1 has been removed. wait_for_volume_removal /usr/local/lib/python2.7/dist-packages/os_brick/initiator/linuxscsi.py:78
2016-02-07 17:21:02.015 DEBUG os_brick.initiator.linuxscsi [req-7330c95a-1491-46ca-85a8-21710375fac2 admin admin] SCSI volume /dev/disk/by-path/ip-192.168.104.10:3260-iscsi-qn.2010-10.org.openstack:volume-1e7f6bd7-cell-4a73-b95e-aabd65a5b188-lun-1 has been removed. wait_for_volume_removal /usr/local/lib/python2.7/dist-packages/os_brick/initiator/linuxscsi.py:84
```

图 8-86

通过 iscsiadm 的 logout, delete 操作断开与 iSCSI target 的连接, 如图 8-87 所示。

```
2016-02-07 17:21:02.546 DEBUG oslo_concurrency.processutils [req-7330c95a-1491-46ca-85a8-21710375fac2 admin admin] CMD "iscsiadm -m node -T qn.2010-10.org.openstack:volume-1e7f6bd7-cell-4a73-b95e-aabd65a5b188 -p 192.168.104.10:3260 --logout" returned: 0 in 0.517s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
2016-02-07 17:21:02.548 DEBUG os_brick.initiator.connector [req-7330c95a-1491-46ca-85a8-21710375fac2 admin admin] ('iscsiadm %(iscsi_command)s: stdout=%(out)s stderr=%(err)s', {'iscsi_command': ('--logout'), 'err': 'Logout of session [sid: 1, target: qn.2010-10.org.openstack:volume-1e7f6bd7-cell-4a73-b95e-aabd65a5b188, portal: 192.168.104.10, 3260] successful.', 'n'}) _run_iscsiadm /usr/local/lib/python2.7/dist-packages/os_brick/initiator/connector.py:658
2016-02-07 17:21:02.548 DEBUG oslo_concurrency.processutils [req-7330c95a-1491-46ca-85a8-21710375fac2 admin admin] Running cmd (subprocess): iscsiadm -m node -T qn.2010-10.org.openstack:volume-1e7f6bd7-cell-4a73-b95e-aabd65a5b188 -p 192.168.104.10:3260 --op delete execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-02-07 17:21:02.581 DEBUG oslo_concurrency.processutils [req-7330c95a-1491-46ca-85a8-21710375fac2 admin admin] CMD "iscsiadm -m node -T qn.2010-10.org.openstack:volume-1e7f6bd7-cell-4a73-b95e-aabd65a5b188 -p 192.168.104.10:3260 --op delete" returned: 0 in 0.033s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
2016-02-07 17:21:02.582 DEBUG os_brick.initiator.connector [req-7330c95a-1491-46ca-85a8-21710375fac2 admin admin] ('iscsiadm %(iscsi_command)s: stdout=%(out)s stderr=%(err)s', {'iscsi_command': ('--op', 'delete'), 'err': 'Error: '}) _run_iscsiadm /usr/local/lib/python2.7/dist-packages/os_brick/initiator/connector.py:658
2016-02-07 17:21:02.582 DEBUG oslo_concurrency.lockutils [req-7330c95a-1491-46ca-85a8-21710375fac2 admin admin] Lock "connect_volume" released by "os_brick.initiator.connector.disconnect_volume" :: held 0.620s inner /usr/local/lib/python2.7/dist-packages/oslo_concurrency/lockutils.py:265
2016-02-07 17:21:02.583 DEBUG nova.virt.libvirt.volume.iscsi [req-7330c95a-1491-46ca-85a8-21710375fac2 admin admin] Disconnected iSCSI volume vdb disconnect_volume /opt/stack/nova/nova/virt/libvirt/volume/iscsi.py:94
```

图 8-87

compue-nova 完成了 detach 工作, 接下来 cinder-volume 就可以删除 volume 相关的 target 了。

#### (4) cinder-volume 删除 target

存储节点 cinder-volume 通过 tgt-admin 命令删除 volume 对应的 target。日志文件为 /opt/stack/logs/c-vol.log, 如图 8-88 所示。

```

2016-02-07 17:21:02.026 INFO cinder.volume.targets.tgt [req-546b875f-4d09-41e8-ab13-065a6d6a301a admin] Removing iscsi target for volume ID: 1e7f6bd7-ce11-4a73-b95e-aabd65a5b188
2016-02-07 17:21:02.027 DEBUG oslo_concurrency.processutils [req-546b875f-4d09-41e8-ab13-065a6d6a301a admin] Running cmd (subprocess): tgt-admin --force --delete ign.2010-10.org.openstack.volume-1e7f6bd7-ce11-4a73-b95e-aabd65a5b188
2016-02-07 17:21:02.102 DEBUG oslo_concurrency.processutils [req-546b875f-4d09-41e8-ab13-065a6d6a301a admin] CMD: tgt-admin --force --delete ign.2010-10.org.openstack.volume-1e7f6bd7-ce11-4a73-b95e-aabd65a5b188 returned: 0 in 0.075s
2016-02-07 17:21:02.103 DEBUG oslo_concurrency.processutils [req-546b875f-4d09-41e8-ab13-065a6d6a301a admin] Running cmd (subprocess): tgt-admin --show execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-02-07 17:21:02.170 DEBUG oslo_concurrency.processutils [req-546b875f-4d09-41e8-ab13-065a6d6a301a admin] CMD: tgt-admin --show returned: 0 in 0.068s
2016-02-07 17:21:02.282 INFO cinder.volume.manager [req-546b875f-4d09-41e8-ab13-065a6d6a301a admin] [volume-1e7f6bd7-ce11-4a73-b95e-aabd65a5b188] Detach volume completed successfully.

```

图 8-88

至此 detach volume 操作已经完成, GUI 也会更新 volume 的 attach 信息, 如图 8-89 所示。

<input type="checkbox"/>	Name	Description	Size	Status	Type	Attached To
<input type="checkbox"/>	vol-1	-	1GB	Available	lvmdriver-1	

图 8-89

## 5. Extend

Extend 操作用于扩大 Volume 的容量, 状态为 Available 的 volume 才能够被 extend。如果 volume 当前已经 attach 给 instance, 需要先 detach 后才能 extend。为了保护现有数据, cinder 不允许缩小 volume。

Extend 实现比较简单, 流程图如 8-90 所示。

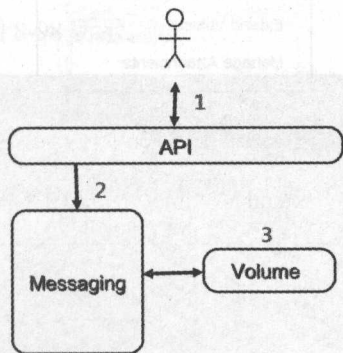


图 8-90

- 向 cinder-api 发送 extend 请求。
- cinder-api 发送消息。
- cinder-volume 执行 extend 操作。

下面我们详细讨论每一个步骤。

(1) 向 cinder-api 发送 extend 请求

客户(可以是 OpenStack 最终用户,也可以是其他程序)向 cinder-api 发送请求:“请 extend 指定的 volume”。

这里我们将 extend volume “vol-2”。

进入 GUI, 操作菜单 Project → Compute → Volumes, 如图 8-91 所示。

<input type="checkbox"/>	Name	Description	Size	Status	Type
<input type="checkbox"/>	vol-2	-	1GB	Available	lvmdriver-1

图 8-91

vol-2 当前大小为 1GB。其在存储节点上对应的 LV 信息如图 8-92 所示。

```
--- Logical volume ---
LV Path                /dev/stack-volumes-lvmdriver-1/volume-c88079bf-
LV Name                 volume-c88079bf-ee48-4b30-beae-c53126ea7ad6
VG Name                 stack-volumes-lvmdriver-1
LV UUID                 gON7Kn-w64s-ViCl-MbYl-QR41-a8nc-Lp7n1D
LV write Access         read/write
LV Creation host, time devstack-controller, 2016-02-08 16:53:24 +0800
LV Status                available
# open                  0
LV Size                 1.00 GiB
Current LE              256
Segments               1
Allocation              inherit
Read ahead sectors      auto
- currently set to      256
Block device            252:3
```

图 8-92

选择 volume “vol-2”, 单击 “Extend Volume”, 如图 8-93 所示。

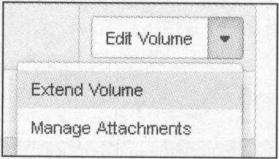


图 8-93

指定新的容量为 3GB, 单击 “Extend Volume”, 如图 8-94 所示。

Extend Volume

Volume Name

vol-2

Current Size (GB)

1

New Size (GB) \*

3

Description:

Extend the size of a volume.

Volume Limits

Total Gigabytes (2 GB) 1,000 GB / available

Cancel

Extend Volume

图 8-94



cinder-api 将接收到 extend volume 的请求。日志文件在 /opt/stack/logs/c-api.log 中,如图 8-95 所示。

```
2016-02-08 16:58:41.293 INFO cinder.api.openstack.wsgi [req-1a6354ea-183e-4476-9854-acf1d0133828 admin] POST http://192.168.104.10:8776/v2/aa81b851d2a54484b6f3984ab2c5d4e4/volumes/c88079bf-ee48-4b30-beae-c53126ea7ad6/action
2016-02-08 16:58:41.293 DEBUG cinder.api.openstack.wsgi [req-1a6354ea-183e-4476-9854-acf1d0133828 admin] Action body: {"os-extend": {"new_size": 3}}
}} get_method /opt/stack/cinder/cinder/api/openstack/wsgi.py:1088
```

图 8-95

## (2) cinder-api 发送消息

cinder-api 发送 extend 消息。cinder-api 没有打印发送消息的日志,只能通过源代码 /opt/stack/cinder/cinder/volume/api.py 查看,方法为 extend,如图 8-96 所示。

```
1263     self.update(context, volume, {'status': 'extending'})
1264     self.volume_rpcapi.extend_volume(context, volume, new_size,
1265                                     reservations)
1266     LOG.info(_LI(
1267         'resource=volume'),
```

图 8-96

## (3) cinder-volume extend volume

cinder-volume 执行 lvextend 命令 extend volume,日志为 /opt/stack/logs/c-vol.log,如图 8-97 所示。

```
2016-02-08 16:58:41.880 DEBUG oslo_concurrency.processutils [req-1a6354ea-183e-4476-9854-acf1d0133828 admin] CMD "lvextend -L 3g stack-volumes-lvm-driver-1/volume-c88079bf-ee48-4b30-beae-c53126ea7ad6" returned: 0 in 0.073s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
2016-02-08 16:58:42.069 INFO cinder.volume.manager [req-1a6354ea-183e-4476-9854-acf1d0133828 admin] [volume-c88079bf-ee48-4b30-beae-c53126ea7ad6] Extend volume completed successfully.
```

图 8-97

LV 被 extend 到 3GB,如图 8-98 所示。

```
-- Logical volume --
LV Path                /dev/stack-volumes-lvm-driver-1/volume-c88079bf-
LV Name                 volume-c88079bf-ee48-4b30-beae-c53126ea7ad6
VG Name                 stack-volumes-lvm-driver-1
LV UUID                 gON7Kn-w64s-V1C1-MDyl-QR41-aBnc-Lp7n1D
LV write Access         read/write
LV Creation host, time devstack-controller, 2016-02-08 16:53:24 +0800
LV Status                available
# open                  0
LV Size                 3.00 GiB
Current LE              768
Segments                1
Allocation               inherit
Read ahead sectors      auto
- currently set to      256
Block device            252:3
```

图 8-98

Extend 操作完成后,GUI 也会更新 volume 的状态信息,如图 8-99 所示。

<input type="checkbox"/>	Name	Description	Size	Status	Type
<input type="checkbox"/>	vol-2	-	3GB	Available	lvmdriver-1

图 8-99

6. Delete

状态为 Available 的 volume 才能够被 delete。如果 volume 当前已经 attach 到 instance，需要先 detach 后才能 delete。

Delete 操作实现比较简单，流程图如图 8-100 所示。

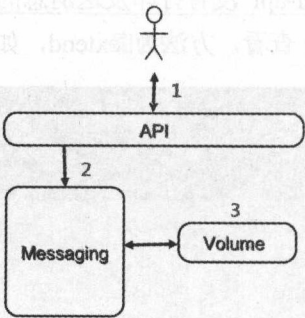


图 8-100

- 向 cinder-api 发送 delete 请求。
- cinder-api 发送消息。
- cinder-volume 执行 delete 操作。

下面我们详细讨论每一个步骤。

(1) 向 cinder-api 发送 delete 请求

客户（可以是 OpenStack 最终用户，也可以是其他程序）向 cinder-api 发送请求：“请 delete 指定的 volume”。

这里我们将 delete volume “vol-2”。

进入 GUI 操作菜单 Project→Compute→Volumes，如图 8-101 所示。

<input type="checkbox"/>	Name	Description	Size	Status	Type
<input type="checkbox"/>	vol-2	-	3GB	Available	lvmdriver-1

图 8-101

选择 volume “vol-2”，单击 “Delete Volume”，如图 8-102 所示。

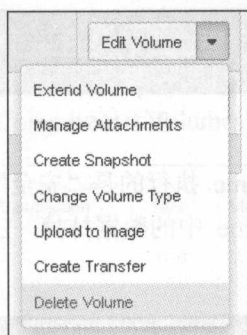


图 8-102

再次确认，如图 8-103 所示。

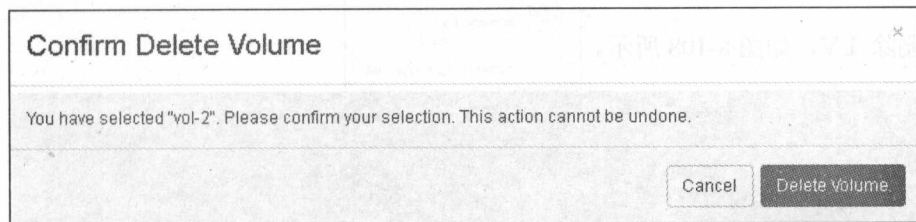


图 8-103

cinder-api 将接收到 delete volume 的请求。日志文件为 /opt/stack/logs/c-api.log，日志内容如图 8-104 所示。

```
2016-02-08 17:28:02.546 INFO cinder.api.openstack.wsgi [req-41ab0f5e-e208-462e-bc8a-3f109d9632f9 admin] DELETE http://192.168.104.10:8776/v2/aa81b851d2a54484b6f3984ab2c5d4e4/volumes/c88079bf-ee48-4b30-beae-c53126ea7ad6
2016-02-08 17:28:02.547 DEBUG cinder.api.openstack.wsgi [req-41ab0f5e-e208-462e-bc8a-3f109d9632f9 admin] Empty body provided in request get_body /opt/stack/cinder/cinder/api/openstack/wsgi.py:862
2016-02-08 17:28:02.547 INFO cinder.api.v2.volumes [req-41ab0f5e-e208-462e-bc8a-3f109d9632f9 admin] Delete volume with id: c88079bf-ee48-4b30-beae-c53126ea7ad6
```

图 8-104

### (2) cinder-api 发送消息

cinder-api 发送 delete 消息。cinder-api 没有打印发送消息的日志，只能通过源代码查看 /opt/stack/cinder/cinder/volume/api.py，方法为 delete，如图 8-105 所示。

```
419
420     self.volume_rpcapi.delete_volume(context, volume, unmanage_
nly)
421     LOG.info(_LI(
422         resource=vref
423     ),
```

图 8-105

### (3) cinder-volume delete volume

cinder-volume 执行 lvremove 命令 delete volume。

日志为 /opt/stack/logs/c-vol.log，内容如图 8-106 所示。

```
2016-02-08 17:28:02.905 INFO cinder.volume.utils [req-41ab0f5e-e208-462e-bc8a-3f109d9632f9 admin] Performing secure delete on volume: /dev/mapper/stack--volumes--lvmdriver--l-volume--c88079bf-ee48-4b30-beae-c53126ea7ad6
```

图 8-106

这里比较有意思的是：cinder-volume 执行的是“安全”删除。

所谓“安全”实际上就是将 volume 中的数据抹掉，LVM driver 使用的是 dd 操作将 LV 的数据清零。日志如图 8-107 所示。

```
2016-02-08 17:28:02.932 DEBUG oslo_concurrency.processutils [req-41ab0f5e-e208-462e-bc8a-3f109d9632f9 admin] Running cmd (subprocess): dd if=/dev/zero of=/dev/mapper/stack--volumes--lvmdriver--l-volume--c88079bf-ee48-4b30-beae-c53126ea7ad6 count=3072 bs=1M oflag=direct execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
```

图 8-107

然后删除 LV，如图 8-108 所示。

```
2016-02-08 17:28:11.628 DEBUG oslo_concurrency.processutils [req-41ab0f5e-e208-462e-bc8a-3f109d9632f9 admin] CMD: "lvremove --config activation {retry_deactivation = 1} -f stack-volumes-lvmdriver-l/volume-c88079bf-ee48-4b30-beae-c53126ea7ad6" returned: 0 in 0.078s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
2016-02-08 17:28:11.629 INFO cinder.volume.drivers.lvm [req-41ab0f5e-e208-462e-bc8a-3f109d9632f9 admin] Successfully deleted volume: c88079bf-ee48-4b30-beae-c53126ea7ad6
```

图 8-108

7. Snapshot

Snapshot 可以为 volume 创建快照，快照中保存了 volume 当前的状态，以后可以通过 snapshot 回溯。snapshot 操作实现比较简单，流程图如图 8-109 所示。

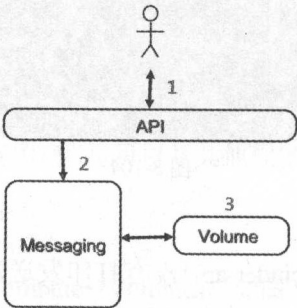


图 8-109

- 向 cinder-api 发送 snapshot 请求。
- cinder-api 发送消息。
- cinder-volume 执行 snapshot 操作。

下面我们详细讨论每一个步骤。

(1) 向 cinder-api 发送 snapshot 请求

客户（可以是 OpenStack 最终用户，也可以是其他程序）向 cinder-api 发送请求：“请



snapshot 指定的 volume”。

这里我们将 snapshot volume “vol-1”。

进入 GUI 操作菜单 Project → Compute → Volumes，如图 8-110 所示。

<input type="checkbox"/>	Name	Description	Size	Status	Type
<input type="checkbox"/>	vol-1	-	1 GB	In-use	lvmdriver-1
Displaying 1 item					

图 8-110

选择 volume “vol-1”，单击“Create Snapshot”，如图 8-111 所示。



图 8-111

为 snapshot 命名，如图 8-112 所示。

Create Volume Snapshot

This volume is currently attached to an instance. In some cases, creating a snapshot from an attached volume can result in a corrupted snapshot.

**Description:**

Volumes are block devices that can be attached to instances.

**Volume Type Description:**

**Snapshot Limits**

Total Gigabytes (1 GB) 1,000 GB Available

Number of Snapshots (0) 10 Available

**Snapshot Name \***

vol-1-snapshot

**Description**

Cancel Create Volume Snapshot (if drive)

图 8-112

这里我们看到界面提示当前 volume 已经 attach 到某个 instance，创建 snapshot 可能导致数据不一致。我们可以先 pause instance，或者确认当前 instance 没有大量的磁盘 IO，处于相对稳定的状态，则可以创建 snapshot，否则还是建议先 detach volume 再做 sanpsnot。

cinder-api 将接收到 snapshot volume 的请求。日志文件查看 /opt/stack/logs/c-api.log，日志如图 8-113 所示。

```

2016-02-08 17:56:11.274 INFO cinder.api.openstack.wsgi [req-66e9fdb7-7d43-4a87-8ea6-1d8897a9f3bd admin] POST http://192.168.104.10:8776/v2/aa81b851d2a54484b6f3984ab2c5d4e4/snapshots
2016-02-08 17:56:11.321 INFO cinder.volume.api [req-66e9fdb7-7d43-4a87-8ea6-1d8897a9f3bd admin] [volume-1e7f6bd7-ce11-4a73-b95e-aabd65a5b188] volume info retrieved successfully.
2016-02-08 17:56:11.322 INFO cinder.api.v2.snapshots [req-66e9fdb7-7d43-4a87-8ea6-1d8897a9f3bd admin] Create snapshot from volume 1e7f6bd7-ce11-4a73-b95e-aabd65a5b188

```

图 8-113

### (2) cinder-api 发送消息

cinder-api 发送消息 snapshot 消息。cinder-api 没有打印发送消息的日志，只能通过源代码查看 `/opt/stack/cinder/cinder/volume/api.py`，方法为 `create_snapshot`。如图 8-114 所示。

```

695     def create_snapshot(self, context,
696                        volume, name, description,
697                        force=False, metadata=None,
698                        cgsnapshot_id=None):
699         snapshot = self.create_snapshot_in_db(
700             context, volume, name,
701             description, force, metadata, cgsnapshot_id)
702         self.volume_rpcapi.create_snapshot(context, volume, snapshot)
703     )
704     return snapshot

```

图 8-114

### (3) cinder-volume 执行 snapshot 操作

cinder-volume 执行 `lvcreate` 命令创建 snapshot。

日志为 `/opt/stack/logs/c-vol.log`，内容如图 8-115 所示。

```

2016-02-08 17:56:11.781 DEBUG oslo_concurrency.processutils [req-66e9fdb7-7d43-4a87-8ea6-1d8897a9f3bd admin] CMD ["lvcreate --name _snapshot-01bd4a4a-a74a-411c-a534-e7532f75e498 --snapshot stack-volumes-lvmdriver-1/volume-1e7f6bd7-ce11-4a73-b95e-aabd65a5b188 -L 1.00g"] returned: 0 in 0.274s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
2016-02-08 17:56:11.894 DEBUG object [req-66e9fdb7-7d43-4a87-8ea6-1d8897a9f3bd admin] cinder object Snapshot has no attribute named: type get /opt/stack/cinder/cinder/objects/base.py:100
2016-02-08 17:56:11.895 INFO cinder.volume.manager [req-66e9fdb7-7d43-4a87-8ea6-1d8897a9f3bd admin] Snapshot(cgsnapshot_id=None, created_at=2016-02-08T09:56:11Z, deleted=False, deleted_at=None, display_description='', display_name='vol-1-snapshot', encryption_key_id=None, id=01bd4a4a-a74a-411c-a534-e7532f75e498, metadata={}, progress='100%', project_id=aa81b851d2a54484b6f3984ab2c5d4e4, provider_auth=None, provider_id=None, provider_location=None, status='available', updated_at=None, user_id=b02f995f9b734384a5662c8d30d661e4, volume=volume(1e7f6bd7-ce11-4a73-b95e-aabd65a5b188), volume_id=1e7f6bd7-ce11-4a73-b95e-aabd65a5b188, volume_size=1, volume_type_id=7cd531f0-4fa2-4a2c-9582-4564cb4f54c3) Create snapshot completed successfully

```

图 8-115

对于 LVM volume provider, snapshot 实际上也是一个 LV，同时记录了与源 LV 的 snapshot 关系，可以通过 `lvdisplay` 查看，如图 8-116 所示。

```

--- Logical volume ---
LV Path                /dev/stack-volumes-lvmdriver-1/_snapshot-01bd4a4
a-a74a-411c-a534-e7532f75e498
LV Name                _snapshot-01bd4a4a-a74a-411c-a534-e7532f75e498
VG Name                stack-volumes-lvmdriver-1
LV UUID                2hyvqp-Fw1i-X45X-i6Nr-yno1-hiEy-opQzt1
LV Write Access        read/write
LV Creation host, time devstack-controller, 2016-02-08 17:56:11 +0800
LV Snapshot status     active destination for volume-1e7f6bd7-ce11-4a73
-b95e-aabd65a5b188
LV Status              available
# open                 0
LV Size                1.00 GiB
Current LE             256
COW-table size         1.00 GiB
COW-table LE           256
Allocated to snapshot  0.00%
Snapshot chunk size    4.00 KiB
Segments               1
Allocation             inherit
Read ahead sectors     auto
- currently set to     256
Block device           252:3

```

图 8-116

GUI 的 Volume Snapshots 标签中可以看到新创建的“vol-1-snapshot”，如图 8-117 所示。

Overview

Instances

Volumes

Images

Volumes

Volume Snapshots

<input type="checkbox"/>	Name	Description	Size
<input type="checkbox"/>	vol-1-snapshot	-	1GB

Displaying 1 item

图 8-117

有了 snapshot，我们就可以将 volume 回溯到创建 snapshot 时的状态。方法是通过 snapshot 创建新的 volume，如图 8-118 所示。

<input type="checkbox"/>	Name	Description	Size	Status	Volume Name	Actions
<input type="checkbox"/>	vol-1-snapshot	-	1GB	Available	vol-1	Create Volume
Displaying 1 item						

图 8-118

新创建的 volume 容量必须大于或等于 snapshot 的容量，如图 8-119 所示。

图 8-119

其过程与 Create Volume 类似，不同之处在于 LV 创建之后会通过 dd 将 snapshot 的数据 copy 到新的 volume。日志分析不再赘述，留个大家练习。

如果一个 volume 存在 snapshot，则这个 volume 是无法删除的。这是因为 snapshot 依赖于 volume，snapshot 无法独立存在。

在 LVM 作为 volume provider 的环境中，snapshot 是从源 volume 完全 copy 而来，所以这种依赖关系不强。但在其他 volume provider（比如商业存储设备或者分布式文件系统），snapshot 通常是源 volume 创建快照时数据状态的一个引用（指针），占用空间非常小，在这种实现方式里 snapshot 对源 volume 的依赖就非常明显了。

## 8. Backup

本节我们讨论 volume 的 Backup 操作。Backup 是将 volume 备份到别的地方（备份设备），将来可以通过 restore 操作恢复。

### （1）Backup VS Snapshot

初看 backup 功能好像与 snapshot 很相似，都可以保存 volume 的当前状态，以备以后恢复。但二者在用途和实现上还是有区别的，具体表现在：

- Snapshot 依赖于源 volume，不能独立存在；而 backup 不依赖源 volume，即便源 volume 不存在了，也可以 restore。
- Snapshot 与源 volume 通常存放在一起，都由同一个 volume provider 管理；而 backup 存放在独立的备份设备中，有自己的备份方案和实现，与 volume provider 没有关系。
- 上面两点决定了 backup 具有容灾功能；而 snapshot 则提供 volume provider 内便捷的回溯功能。

### （2）配置 cinder-backup

Cinder 的 backup 功能是由 cinder-backup 服务提供的，devstack 默认没有启用该服务，需要手工启用。与 cinder-volume 类似，cinder-backup 也通过 driver 架构支持多种备份 backend，



包括 POSIX 文件系统、NFS、Ceph、GlusterFS、Swift 和 IBM TSM。支持的 driver 源文件放在 `/opt/stack/cinder/cinder/backup/drivers/`。如图 8-120 所示。

```
root@devstack-controller:~# ls -l /opt/stack/cinder/cinder/backup/drivers/*.py
-rw-r--r-- 1 stack stack 0 Dec 10 20:17 /opt/stack/cinder/cinder/backup/drivers/__init__.py
-rw-r--r-- 1 stack stack 49789 Dec 10 20:17 /opt/stack/cinder/cinder/backup/drivers/ceph.py
-rw-r--r-- 1 stack stack 3456 Dec 10 20:17 /opt/stack/cinder/cinder/backup/drivers/glusterfs.py
-rw-r--r-- 1 stack stack 3112 Dec 10 20:17 /opt/stack/cinder/cinder/backup/drivers/nfs.py
-rw-r--r-- 1 stack stack 5361 Dec 10 20:17 /opt/stack/cinder/cinder/backup/drivers/posix.py
-rw-r--r-- 1 stack stack 13057 Dec 10 20:17 /opt/stack/cinder/cinder/backup/drivers/swift.py
-rw-r--r-- 1 stack stack 20822 Dec 10 20:17 /opt/stack/cinder/cinder/backup/drivers/tsm.py
```

图 8-120

本节我们将以 NFS 为 backend 来研究 backup 操作。

在实验环境中, 存放 volume backup 的 NFS 远程目录为 `192.168.104.11:/backup`

cinder-backup 服务节点上 mount point 为 `/backup_mount`。

需要在 `/etc/cinder/cinder.conf` 中做相应的配置, 如图 8-121 所示。

```
[DEFAULT]
backup_driver = cinder.backup.drivers.nfs
backup_mount_point_base = /backup_mount
backup_share = 192.168.104.11:/backup
```

图 8-121

然后手工启动 cinder-backup 服务。

```
/usr/bin/python /usr/local/bin/cinder-backup --config-file
/etc/cinder/cinder.conf
```

一切准备就绪, 下面我们来看 backup 操作的流程, 如图 8-122。

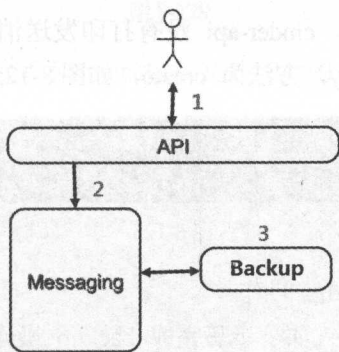


图 8-122

- 向 cinder-api 发送 backup 请求。
- cinder-api 发送消息。
- cinder-backup 执行 backup 操作。

下面我们详细讨论每一个步骤。

(3) 向 cinder-api 发送 backup 请求

客户(可以是 OpenStack 最终用户,也可以是其他程序)向 cinder-api 发送请求:“请 backup 指定的 volume。”

这里我们将 backup volume “vol-1”, 目前 backup 只能在 CLI 中执行, 如图 8-123 所示。

```
root@devstack-controller:~# cinder backup-create vol-1 --force
+-----+-----+
| Property | Value |
+-----+-----+
| id        | a68c9eb8-be32-4192-9278-e4778beee59e |
| name      | None |
| volume_id | 1e7f6bd7-ce11-4a73-b95e-aabd65a5b188 |
+-----+-----+
```

图 8-123

这里因为 vol-1 已经 attach 到 instance, 需要使用 --force 选项。

cinder-api 接收到 backup volume 的请求。日志文件查看 /opt/stack/logs/c-api.log, 日志内容如图 8-124 所示。

```
2016-02-09 18:35:34.256 INFO cinder.api.openstack.wsgi [req-344040f8-b2c5-4d71-9f67-1367208ec381 admin] POST http://192.168.104.10:8776/v2/aa81b851d2a54484b6f3984ab2c5d4e4/backups
2016-02-09 18:35:34.257 DEBUG cinder.api.contrib.backups [req-344040f8-b2c5-4d71-9f67-1367208ec381 admin] Creating new backup {'u' backup': {'u' container': None, 'u' description': None, 'u' incremental': False, 'u' volume_id': '1e7f6bd7-ce11-4a73-b95e-aabd65a5b188', 'u' force': True, 'u' name': None}}
2016-02-09 18:35:34.257 INFO cinder.api.contrib.backups [req-344040f8-b2c5-4d71-9f67-1367208ec381 admin] Creating backup of volume 1e7f6bd7-ce11-4a73-b95e-aabd65a5b188 in container None
```

图 8-124

(4) cinder-api 发送消息

cinder-api 发送 backup 消息。cinder-api 没有打印发送消息的日志, 只能通过源代码查看 /opt/stack/cinder/cinder/backup/api.py, 方法为 create, 如图 8-125 所示。

```
256 self.backup_rpcapi.create_backup(context, backup)
257
258 return backup
```

图 8-125

(5) cinder-backup 执行 backup 操作

cinder-backup 收到消息后, 通过如下步骤完成 backup 操作, 日志为 /opt/stack/logs/c-vol.log。启动 backup 操作, mount NFS, 如图 8-126 所示。

```
2016-02-09 18:35:34.602 INFO cinder.backup.manager [req-344040f8-b2c5-4d71-9f67-1367208ec381 admin] Create backup started, backup: a68c9eb8-be32-4192-9278-e4778beee59e volume: 1e7f6bd7-ce11-4a73-b95e-aabd65a5b188.
2016-02-09 18:35:34.613 DEBUG oslo.concurrency.processutils [req-344040f8-b2c5-4d71-9f67-1367208ec381 admin] Running cmd (subprocess): mount -execu te /usr/local/lib/python2.7/dist-packages/oslo.concurrency/processutils.py:250
2016-02-09 18:35:34.623 DEBUG oslo.concurrency.processutils [req-344040f8-b2c5-4d71-9f67-1367208ec381 admin] cmd "mount" returned: 0 in 0.009s execute /usr/local/lib/python2.7/dist-packages/oslo.concurrency/processutils.py:280
2016-02-09 18:35:34.624 INFO os_brick.remotefs.remotefs [req-344040f8-b2c5-4d71-9f67-1367208ec381 admin] Already mounted: /backup_mount/51197615090125a33d354e3ce4023674
2016-02-09 18:35:34.624 DEBUG cinder.backup.drivers.nfs [req-344040f8-b2c5-4d71-9f67-1367208ec381 admin] using NFS backup repository: /backup_mount/51197615090125a33d354e3ce4023674 __init__ /opt/stack/cinder/cinder/backup/drivers/nfs.py:58
```

图 8-126

创建 volume 的临时快照, 如图 8-127 所示。

```
2016-02-09 18:35:34.781 DEBUG oslo_concurrency.processutils [req-344040f8-b2c5-4d71-9f67-1367208ec381 admin] Running cmd (subprocess): lvcreate --name _snapshot-d0d40e8c-4747-4e39-94fa-6a0b945f1359 --snapshot stack-volumes-lvmdriver-1/volume-1e7f6bd7-cell1-4a73-b95e-aabd65a5b188 -L 1.00g execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-02-09 18:35:34.975 DEBUG oslo_concurrency.processutils [req-344040f8-b2c5-4d71-9f67-1367208ec381 admin] CMD "lvcreate --name _snapshot-d0d40e8c-4747-4e39-94fa-6a0b945f1359 --snapshot stack-volumes-lvmdriver-1/volume-1e7f6bd7-cell1-4a73-b95e-aabd65a5b188 -L 1.00g" returned: 0 in 0.194s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
```

图 8-127

创建存放 backup 的 container 目录, 如图 8-128 所示。

```
2016-02-09 18:35:35.125 DEBUG cinder.backup.chunkeddriver [req-344040f8-b2c5-4d71-9f67-1367208ec381 admin] _create_container started, container: a6/8c/a68c9eb8-be32-4192-9278-e4778beee59e, backup: a68c9eb8-be32-4192-9278-e4778beee59e, _create_container /opt/stack/cinder/cinder/backup/chunkeddriver.py:164
```

图 8-128

对临时快照数据进行压缩, 并保存到 container 目录, 如图 8-129 所示。

```
2016-02-09 18:35:35.167 DEBUG cinder.backup.chunkeddriver [req-344040f8-b2c5-4d71-9f67-1367208ec381 admin] starting backup of volume: 1e7f6bd7-cell1-4a73-b95e-aabd65a5b188, volume size: 1073741824, object names prefix backup, availability zone: nova _prepare_backup /opt/stack/cinder/cinder/backup/chunkeddriver.py:285
2016-02-09 18:35:48.096 DEBUG cinder.backup.chunkeddriver [req-344040f8-b2c5-4d71-9f67-1367208ec381 admin] Backing up chunk of data from volume. _backup_chunk /opt/stack/cinder/cinder/backup/chunkeddriver.py:309
2016-02-09 18:35:56.149 DEBUG cinder.backup.chunkeddriver [req-344040f8-b2c5-4d71-9f67-1367208ec381 admin] Compressed 1073741824 bytes of data to 1043644 bytes using zlib. _prepare_output_data /opt/stack/cinder/cinder/backup/chunkeddriver.py:349
```

图 8-129

创建并保存 sha256 (加密) 文件和 metadata 文件, 如图 8-130 所示。

```
2016-02-09 18:35:58.788 DEBUG cinder.backup.chunkeddriver [req-5aaaae41d-0b1a-4925-bfce-686c2f37193e None] _write_sha256file finished. _write_sha256file /opt/stack/cinder/cinder/backup/chunkeddriver.py:231
2016-02-09 18:35:58.789 DEBUG cinder.backup.chunkeddriver [req-5aaaae41d-0b1a-4925-bfce-686c2f37193e None] _write_metadata started, container name: a6/8c/a68c9eb8-be32-4192-9278-e4778beee59e, metadata filename: backup_metadata, _write_metadata /opt/stack/cinder/cinder/backup/chunkeddriver.py:192
2016-02-09 18:35:58.794 DEBUG cinder.backup.chunkeddriver [req-5aaaae41d-0b1a-4925-bfce-686c2f37193e None] _write_metadata finished. Metadata: {
  "backup_description": null,
  "backup_id": "a68c9eb8-be32-4192-9278-e4778beee59e",
  "backup_name": null,
  "created_at": "2016-02-09 10:35:34+00:00",
  "objects": [
    {
      "backup-00001": {
        "compression": "zlib",
        "length": 1073741824,
        "md5": "cd573cfaace07e7949bc0c46028904ff",
        "offset": 0
      }
    }
  ],
  "parent_id": null,
  "version": "1.0.0",
  "volume_id": "1e7f6bd7-cell1-4a73-b95e-aabd65a5b188",
}
```

图 8-130

删除临时快照, 如图 8-131 所示。

```
2016-02-09 18:35:58.895 INFO cinder.volume.utils [req-5aaaae41d-0b1a-4925-bf6e-686c2f37193e None] Performing secure delete on volume: /dev/mapper/s
tack--volumes--lvmdriver--1/_snapshot-d0d40e8c-4747-4e39-94fa--6a0b94
5f1359-cow
2016-02-09 18:36:00.848 DEBUG oslo_concurrency.processutils [req-5aaaae41d
-0b1a-4925-bf6e-686c2f37193e None] Running cmd (subprocess): lvremove --c
onfig activation { retry_deactivation = 1} -f stack-volumes-lvmdriver-1/_
snapshot-d0d40e8c-4747-4e39-94fa-6a0b945f1359 execute /usr/local/lib/pyt
hon2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-02-09 18:36:00.969 DEBUG oslo_concurrency.processutils [req-5aaaae41d
-0b1a-4925-bf6e-686c2f37193e None] CMD "lvremove --config activation { re
try_deactivation = 1} -f stack-volumes-lvmdriver-1/_snapshot-d0d40e8c-47
47-4e39-94fa-6a0b945f1359" returned: 0 in 0.121s execute /usr/local/lib/p
ython2.7/dist-packages/oslo_concurrency/processutils.py:280
2016-02-09 18:36:01.089 INFO cinder.backup.manager [req-5aaaae41d-0b1a-492
5-bf6e-686c2f37193e None] Create backup finished. backup: a68c9eb8-be32-4
192-9278-e4778beee59e.
```

图 8-131

Backup 完成后，我们可以查看一下 container 目录的内容，如图 8-132 所示。

```
root@devstack-controller:~# ls -l /backup_mount/51197615090125a
33d354e3ce4023674/a6/8c/a68c9eb8-be32-4192-9278-e4778beee59e/
total 3364
-rw-rw---- 1 root root 1043644 Feb  9 18:35 backup-00001
-rw-rw---- 1 root root 1834 Feb  9 18:36 backup_metadata
-rw-rw---- 1 root root 2392344 Feb  9 18:36 backup_sha256file
```

图 8-132

里面有三个文件，根据前面的日志我们可以知道：

- backup-00001，压缩后的 backup 文件。
- backup\_metadata，metadata 文件。
- backup\_sha256file，加密文件。

可以通过 cinder backup-list 查看当前存在的 backup，如图 8-133 所示。

```
root@devstack-controller:~# cinder backup-list
+-----+-----+-----+
| ID | Volume ID | Status |
+-----+-----+-----+
| a68c9eb8-be32-4192-9278-e4778beee59e | 1e7f6bd7-cell-4a73-b95e-aabd65a5b188 | available |
+-----+-----+-----+
```

图 8-133

另外，我们可以查看一下 cinder backup-create 的用法，如图 8-134 所示。

```
root@devstack-controller:~# cinder help backup-create
usage: cinder backup-create [--container <container>] [--name <name>]
                             [--description <description>] [--incremental]
                             [--force]
                             <volume>

Creates a volume backup.

Positional arguments:
  <volume>          Name or ID of volume to backup.

Optional arguments:
  --container <container>
                    Backup container name. Default=None.
  --name <name>     Backup name. Default=None.
  --description <description>
                    Backup description. Default=None.
  --incremental
                    Incremental backup. Default=False.
  --force           Allows or disallows backup of a volume when the volume
                    is attached to an instance. If set to true, backs up
                    the volume whether its status is "available" or "in-
                    use". The backup of an "in-use" volume means your data
                    is crash consistent. Default=False.
```

图 8-134

这里有 --incremental 选项，表示可以执行增量备份。



如果之前做过普通（全量）备份，之后可以通过增量备份大大减少需要备份的数据量，是个很不错的功能。增量备份的操作和日志分析留给大家做练习。

9. Restore

本节我们讨论如何 restore volume。

restore 的过程其实很简单，两步走：

- 在存储节点上创建一个空白 volume。
- 将 backup 的数据 copy 到空白 volume 上。

下面我们来看 restore 操作的详细流程，如图 8-135：

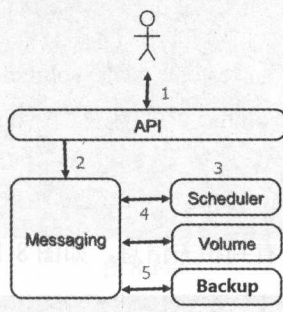


图 8-135

- 向 cinder-api 发送 backup 请求。
- cinder-api 发送消息。
- cinder-scheduler 挑选最合适的 cinder-volume。
- cinder-volume 创建空白 volume。
- cinder-backup 将 backup 数据 copy 到空白 volume 上。

我们先来看第 1 步。

(1) 向 cinder-api 发送 restore 请求

客户（可以是 OpenStack 最终用户，也可以是其他程序）向 cinder-api 发送请求：“请 restore 指定的 backup”。这里我们将 restore 之前创建的 backup。

目前 restore 只能在 CLI 中执行，如图 8-136 所示。

```
root@devstack-controller:~# cinder backup-list
+-----+-----+-----+
| ID | volume ID | status |
+-----+-----+-----+
| a68c9eb8-be32-4192-9278-e4778beee59e | 1e7f6bd7-ce11-4a73-b95e-aabd65a5b188 | available |
+-----+-----+-----+

root@devstack-controller:~# cinder backup-restore a68c9eb8-be32-4192-9278-e4778beee59e
+-----+-----+
| Property | value |
+-----+-----+
| backup_id | a68c9eb8-be32-4192-9278-e4778beee59e |
| volume_id | abcd4191-71c6-4735-b6bd-d2a73592932a |
| volume_name | restore_backup_a68c9eb8-be32-4192-9278-e4778beee59e |
+-----+-----+
```

图 8-136

↑ **cinder-api** 接收到 **restore** 请求。日志文件查看 `/opt/stack/logs/c-api.log`。日志内容如图 8-137 所示。

```
2016-02-09 19:10:59.575 INFO cinder.api.openstack.wsgi [req-fbceb475-ec16-422a-a700-776773ebf12a admin] POST http://192.168.104.10:8776/v2/aa81b851d2a54484b6f3984ab2c5d4e47/backups/a68c9eb8-be32-4192-9278-e4778beee59e/restore
2016-02-09 19:10:59.576 DEBUG cinder.api.contrib.backups [req-fbceb475-ec16-422a-a700-776773ebf12a admin] Restoring backup a68c9eb8-be32-4192-9278-e4778beee59e ({'restore': {'volume_id': None}}) r
restore /opt/stack/cinder/cinder/api/contrib/backups.py:291
2016-02-09 19:10:59.576 INFO cinder.api.contrib.backups [req-fbceb475-ec16-422a-a700-776773ebf12a admin] Restoring backup a68c9eb8-be32-4192-9278-e4778beee59e to volume None
2016-02-09 19:10:59.584 INFO cinder.backup.api [req-fbceb475-ec16-422a-a700-776773ebf12a admin] Creating volume of 1 GB for restore of backup a68c9eb8-be32-4192-9278-e4778beee59e.
```

图 8-137

这里看到 **cinder-api** 转发请求，为 **restore** 创建 **volume**。

之后 **cinder-scheduler** 和 **cinder-volume** 将创建空白 **volume**，这个过程与 **create volume** 一样，不再赘述。

接下来分析数据恢复的过程。

首先，在 **cinder-api** 日志中可以看到相关信息，如图 8-138 所示。

```
2016-02-09 19:11:00.939 INFO cinder.backup.api [req-fbceb475-ec16-422a-a700-776773ebf12a admin] Overwriting volume abcd4191-71c6-4735-b6bd-d2a73592932a with restore of backup a68c9eb8-be32-4192-9278-e4778beee59e
```

图 8-138

这里注意日志中的 **volumeid** 和 **backupid** 与前面 **backup-restore** 命令的输出是一致的。

下面来看 **cinder-backup** 是如何恢复数据的。

## (2) cinder-backup 执行 restore 操作

日志查看 `/opt/stack/logs/c-vol.log`。

启动 **restore** 操作，**mount NFS**，如图 8-139 所示。

```
2016-02-09 19:11:01.007 INFO cinder.backup.manager [req-fbceb475-ec16-422a-a700-776773ebf12a admin] Restore backup started, backup: a68c9eb8-be32-4192-9278-e4778beee59e volume: abcd4191-71c6-4735-b6bd-d2a73592932a.
2016-02-09 19:11:01.071 DEBUG oslo_concurrency.processutils [req-fbceb475-ec16-422a-a700-776773ebf12a admin] Running cmd (subprocess): mount --exec -- /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250.
2016-02-09 19:11:01.079 DEBUG oslo_concurrency.processutils [req-fbceb475-ec16-422a-a700-776773ebf12a admin] CMD "mount" returned: 0 in 0.008s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
2016-02-09 19:11:01.080 INFO os_brick.remotefs.remotefs [req-fbceb475-ec16-422a-a700-776773ebf12a admin] Already mounted: /backup_mount/51197615090125a33d354e3ce4023674
2016-02-09 19:11:01.081 DEBUG cinder.backup.drivers.nfs [req-fbceb475-ec16-422a-a700-776773ebf12a admin] Using NFS backup repository: /backup_mount/51197615090125a33d354e3ce4023674 -- /opt/stack/cinder/cinder/backup/drivers/nfs.py:58
```

图 8-139

读取 container 目录中的 metadata，如图 8-140 所示。

```
2016-02-09 19:11:01.085 DEBUG cinder.backup.chunkeddriver [req-fbc
eb475-ec16-422a-a700-776773ebf12a admin] starting restore of backu
p backup container: a68c/a68c9eb8-be32-4192-9278-e4778beee59e, to
volume abcd4191-71c6-4735-b6bd-d2a73592932a, backup: a68c9eb8-be3
2-4192-9278-e4778beee59e, restore /opt/stack/cinder/cinder/backup
chunkeddriver.py:648
2016-02-09 19:11:01.085 DEBUG cinder.backup.chunkeddriver [req-fbc
eb475-ec16-422a-a700-776773ebf12a admin] _read_metadata started, c
ontainer_name: a68c/a68c9eb8-be32-4192-9278-e4778beee59e, metadat
a filename: backup_metadata, _read_metadata /opt/stack/cinder/cind
er/backup/chunkeddriver.py:238
2016-02-09 19:11:01.088 DEBUG cinder.backup.chunkeddriver [req-fbc
eb475-ec16-422a-a700-776773ebf12a admin] _read_metadata finished.
Metadata:
  "backup_description": null,
  "backup_id": "a68c9eb8-be32-4192-9278-e4778beee59e",
  "backup_name": null,
  "created_at": "2016-02-09 10:35:34+00:00",
  "objects": [
    {
      "backup-00001": {
        "compression": "zlib",
        "length": 1073741824,
        "md5": "cd573cfaace07e7949bc0c46028904ff",
        "offset": 0
      }
    }
  ],
  "parent_id": null,
  "version": "1:0.0",
  "volume_id": "1e/f6bd7-cell-4a73-b95e-aabd65a5b188",
}
```

图 8-140

将数据解压并写到 volume 中，如图 8-141 所示。

```
2016-02-09 19:11:01.094 DEBUG cinder.backup.chunkeddriver [req-fbc
eb475-ec16-422a-a700-776773ebf12a admin] decompressing data using
zlib algorithm _restore_v1 /opt/stack/cinder/cinder/backup/chunked
driver.py:610
```

图 8-141

恢复 volume 的 metadata，完成 restore 操作，如图 8-142 所示。

```
2016-02-09 19:11:06.392 DEBUG cinder.backup.driver [req-fbc475-e
c16-422a-a700-776773ebf12a admin] Restoring volume base metadata _
restore_vol_base_meta /opt/stack/cinder/cinder/backup/driver.py:16
7
2016-02-09 19:11:06.634 INFO cinder.backup.manager [req-fbc475-e
c16-422a-a700-776773ebf12a admin] Restore backup finished, backup
a68c9eb8-be32-4192-9278-e4778beee59e restored to volume abcd4191-7
1c6-4735-b6bd-d2a73592932a.
```

图 8-142

此时，在 GUI 中已经可以看到 restore 创建的 volume，如图 8-143 所示。

<input type="checkbox"/>	Name	Description	Size	Status
<input type="checkbox"/>	restore_backup_a68c9eb8-be32-4192-9278-e4778beee59e	auto-created_from_restore_from_backup	1GB	Available
<input type="checkbox"/>	vol-1		1GB	In-use

图 8-143

10. Boot from Volume

Volume 除了可以用作 instance 的数据盘，也可以作为启动盘（Bootable Volume），那么如何使 volume 成为 bootable 呢？

现在我们打开 instance 的 launch 操作界面，如图 8-144 所示。

Launch Instance

Details \*

Access & Security

Networking \*

Availability Zone

nova

Instance Name \*

Flavor \* ?

m1.nano

Instance Count \* ?

1

Instance Boot Source \* ?

Select source

Select source

Boot from image

Boot from snapshot

Boot from volume

Boot from image (creates a new volume)

Boot from volume snapshot (creates a new volume)

图 8-144

这里有一个下拉菜单“Instance Boot Source”。以前我们 launch instance 要么直接从 image launch（Boot from image），要么从 instance 的 snapshot launch（Boot from snapshot）。

这两种 launch 方式下，instance 的启动盘 vda 均为镜像文件，存放路径为计算节点 /opt/stack/data/nova/instances/<instance\_id>/disk，例如图 8-145 所示。

```
<disk type=...>
<driver name=...>
<source file=...>
<target dev=...>
<address type=...>
</disk>
```

图 8-145

下拉列表的后三项则可以将 volume 作为 instance 的启动盘 vda，分别为：

- Boot from volume



直接从现有的 bootable volume launch。

- Boot from image (create a new volume)

创建一个新的 volume，将 image 的数据 copy 到 volume，然后从该 volume launch。

- Boot from volume snapshot (create a new volume)

通过指定的 volume snapshot 创建 volume，然后从该 volume launch，当然前提是该 snapshot 对应的源 volume 是 bootable 的。

下面我们以 Boot from image (create a new volume) 为例，看如何从 volume 启动，如图 8-146 所示。

图 8-146

选择 cirros 作为 image，instance 命名为“c3”。

如果希望 terminate instant 的时候同时删除 volume，可以选中“Delete on Terminate”。

c3 成功 Launch 后，volume 列表中可以看到一个新 bootable volume，以 volume ID 命名，并且已经 attach 到 c3，如图 8-147 所示。

<input type="checkbox"/>	Name	Description	Size	Status	Type	Attached To	Availability Zone	Bootable
<input type="checkbox"/>	c3defda3-ca0f-495d-a7fa-12f1c0fb91df	-	1GB	In-use	lvmdriver-1	Attached to c3 on /dev/vda	nova	Yes

图 8-147

该 volume 已经配置为 c3 的启动盘 vda，如图 8-148 所示。

```
disk_type=
disk_image_name=
source=
target_dev=
serial=c3defda3-ca0f-495d-a7fa-12f1c0fb91df
address_type=
function=
```

图 8-148

如果用该 volume 创建 snapshot 之后，就可以通过 Boot from volume snapshot (create a new volume) 部署新的 instance，这个操作留给大家练习。

这里再给大家留个练习：boot from volume 的 instance 也可以执行 live migrate，请大家思考一下 volume 是如何 migrate 到目标节点的，并通过日志验证。

11. NFS Volume Provider

cinder-volume 支持多种 volume provider，前面我们一直使用的是默认的 LVM，本节我们将增加 NFS volume provider。

虽然 NFS 更多地应用在实验或小规模 cinder 环境，由于性能和缺乏高可用的原因在生产环境中不太可能使用，但是学习 NFS volume provider 的意义在于：

- 理解 cinder-volume 如何支持多 backend。
- 更重要的，可以理解 cinder-volume、nova-compute 和 volume provider 是如何协同工作，共同为 instance 提供块存储。
- 举一反三，能够快速理解并接入其他生产级 backend，比如 Ceph、商业存储等。

图 8-149 展示了 cinder、nova 是如何与 NFS volume provider 协调工作的。

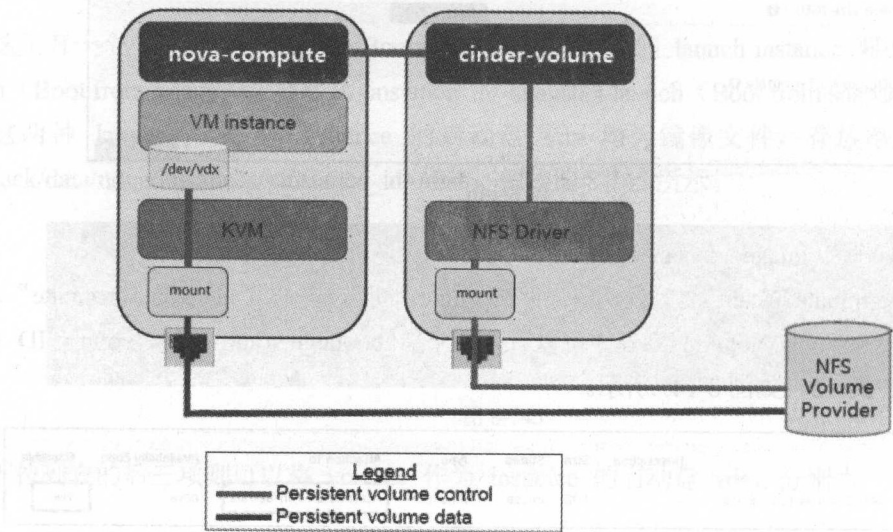


图 8-149

## ● NFS Volume Provider

就是我们通常说的 NFS Server，提供远程 NFS 目录，NFS Client 可以 mount 这些远程目录到本地，然后像使用本地目录一样创建、读写文件以及子目录。

### ● cinder-volume

存储节点通过 NFS driver 管理 NFS volume provider 中的 volume，这些 volume 在 NFS 中实际上是一个个文件。

### ● nova-compute

计算节点将 NFS volume provider 存放 volume 的目录 mount 到本地，然后将 volume 文件作为虚拟硬盘映射给 instance。

这里几点需要强调：

在 Cinder 的 driver 架构中，运行 cinder-volume 的存储节点和 Volume Provider 可以是完全独立的两个实体。cinder-volume 通过 driver 与 Volume Provider 通信，控制和管理 volume。

Instance 读写 volume 时，数据流不需要经过存储节点，而是直接对 Volume Provider 中的 volume 进行读写。

正如图 8-149 所示，存储节点与 NFS Volume Provider 的连接只用作 volume 的管理和控制（绿色连线）。真正的数据读写，是通过计算节点和 NFS Volume Provider 之间的连接完成的（紫色连线）。这种设计减少了中间环节，存储节点不直接参与数据传输，保证了读写效率。

其他 Volume Provider（例如 ceph，swift，商业存储等）均遵循这种控制流与数据流分离的设计。

### (1) 配置 NFS Volume Provider

在实验环境中，NFS volume provider 的 NFS 远程目录为 192.168.104.11:/storage  
cinder-volume 服务节点上 mount point 为 /nfs\_storage。

在 /etc/cinder/cinder.conf 中添加 nfs backend，如图 8-150 所示。

```
enabled_backends = lvmdriver-1,nfs

[lvmdriver-1]
lvm_type = default
iscsi_helper = tgtadm
volume_group = stack-volumes-lvmdriver-1
volume_driver = cinder.volume.drivers.lvm.LVMVolumeDriver
volume_backend_name = lvmdriver-1

[nfs]
nfs_mount_point_base = /nfs_storage
nfs_shares_config = /etc/cinder/nfs_shares
volume_driver=cinder.volume.drivers.nfs.NfsDriver
volume_backend_name = nfs
```

图 8-150

enabled\_backends = lvmdriver-1,nfs 让 cinder-volume 使用 nfs backend

[nfs] 中详细配置 nfs backend。包括：

① 指定存储节点上 `/nfs_storage` 为 `nfs` 的 `mount point`。

```
nfs_mount_point_base = /nfs_storage
```

② 查看 `/etc/cinder/nfs_shares` 活动 `nfs` 共享目录列表。

其内容为 `nfs_shares_config = /etc/cinder/nfs_shares`，如图 8-151 所示。

```
root@devstack-controller:~# cat /etc/cinder/nfs_shares
192.168.104.11:/storage
root@devstack-controller:~#
```

图 8-151

列表中只有 `192.168.104.11:/storage`。如果希望有多个 `nfs` 共享目录存放 `volume`，则可以添加到该文件中。

③ `nfs volume driver`。

```
volume_driver=cinder.volume.drivers.nfs.NfsDriver
```

④ 设置 `volume backend name`。在 `cinder` 中需要根据这里的 `volume_backend_name` 创建对应的 `volume type`，这个非常重要。

```
volume_backend_name = nfs
```

重启 `cinder-volume`，`cinder service-list` 确认 `nfs cinder-volume` 服务正常工作，如图 8-152 所示。

```
root@devstack-controller:~# cinder service-list
```

Binary	Host	Zone	Status	State
cinder-backup	devstack-controller	nova	enabled	up
cinder-scheduler	devstack-controller	nova	enabled	up
cinder-volume	devstack-controller@lvmdriver-1	nova	enabled	up
cinder-volume	devstack-controller@nfs	nova	enabled	up

图 8-152

创建 `nfs volume type`。

打开 GUI 页面，执行 `Admin → System → Volumes → Volume Types`，单击“`Create Volume Type`”，如图 8-153 所示。

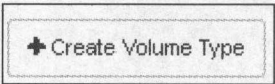
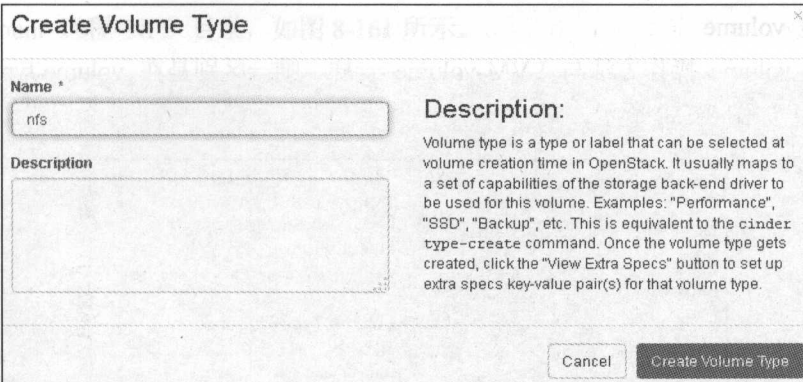


图 8-153

命名 `nfs`，单击“`Create Volume Type`”，如图 8-154 所示。





**Create Volume Type**

**Name \***  
nfs

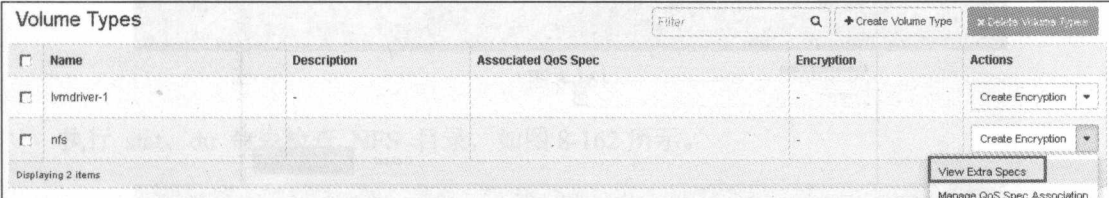
**Description**

**Description:**  
Volume type is a type or label that can be selected at volume creation time in OpenStack. It usually maps to a set of capabilities of the storage back-end driver to be used for this volume. Examples: "Performance", "SSD", "Backup", etc. This is equivalent to the cinder type-create command. Once the volume type gets created, click the "View Extra Specs" button to set up extra specs key-value pair(s) for that volume type.

Cancel Create Volume Type

图 8-154

选择 nfs volume type，单击下拉菜单“View Extra Specs”，如图 8-155 所示。



Name	Description	Associated QoS Spec	Encryption	Actions
lvmdriver-1			-	Create Encryption
nfs			-	Create Encryption View Extra Specs Manage QoS Spec Association

Displaying 2 items

图 8-155

单击“Create”，Key 输入 volume\_backend\_name，Value 输入 nfs，如图 8-156 所示。



**Create Volume Type Extra Spec**

**Key \***  
volume\_backend\_name

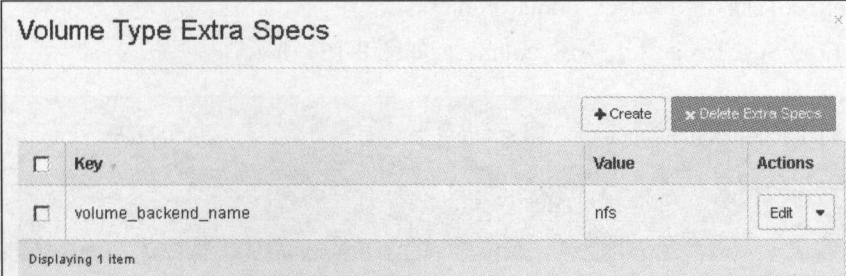
**Value \***  
nfs

**Description:**  
Create a new "extra spec" key-value pair for a volume type.

Cancel Create

图 8-156

NFS volume provider 准备就绪，可以创建 volume 了，如图 8-157 所示。



Key	Value	Actions
volume_backend_name	nfs	Edit

Displaying 1 item

图 8-157

## (2) 创建 volume

创建 NFS volume 操作方法与 LVM volume 一样，唯一区别是在 volume type 的下拉列表中选择“nfs”，如图 8-158 所示。

图 8-158

单击“Create Volume”，cinder-api、cinder-scheduler 和 cinder-volume 共同协作创建 volume “nfs-vol-1”。这个流程与 LVM volume 一样。

下面我们重点分析 cinder-volume 的日志，看看 NFS volume provider 是如何创建 volume 的。

查看日志/opt/stack/logs/c-vol.log。cinder-volume 也会启动 Flow 来完成 volume 创建工作，Flow 的名称为 volume\_create\_manager，如图 8-159 所示。

```
2016-02-10 18:58:59.452 DEBUG cinder.volume.manager [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] Flow 'volume_create_manager' (7ble7234-4ab8-409b-91d4-4fa848c3a66e) transitioned into state 'RUNNING' from state 'PENDING' _flow_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:140
```

图 8-159

volume\_create\_manager 首先执行 ExtractVolumeRefTask、OnFailureRescheduleTask、ExtractVolumeSpecTask，NotifyVolumeActionTask 为 volume 创建做准备。然后由 CreateVolumeFromSpecTask 真正创建 volume，如图 8-160 所示。

```
2016-02-10 18:58:59.845 DEBUG cinder.volume.manager [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] Task 'cinder.volume.flows.manager.create_volume.create_volume_from_spec_task;volume:create' (51f821d7-592a-40ba-9421-3c93d85a6517) transitioned into state 'RUNNING' from state 'PENDING' _task_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:189
2016-02-10 18:58:59.846 INFO cinder.volume.flows.manager.create_volume [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] volume 7a92a9dd-d1c5-43ef-929a-c00e9a4fd576: being created as raw with specification: {'status': 'u'creating', 'volume_size': '1', 'volume_name': 'u'volume-7a92a9dd-d1c5-43ef-929a-c00e9a4fd576'}
```

图 8-160

首先 mount 远程 NFS 目录, 如图 8-161 所示。

```
2016-02-10 18:58:59.847 DEBUG cinder.volume.drivers.remotefs [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] Loading shares from /etc/cinder/nfs_shares. _load_shares_config /opt/stack/cinder/cinder/volume/drivers/remotefs.py:459
2016-02-10 18:58:59.847 DEBUG cinder.volume.drivers.remotefs [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] shares loaded: {'192.168.104.11:/storage': None} _load_shares_config /opt/stack/cinder/cinder/volume/drivers/remotefs.py:491
2016-02-10 18:58:59.847 DEBUG oslo_concurrency.processutils [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] Running cmd (subprocess): mount -t nfs -o rw,fsid=192.168.104.11:/storage /opt/stack/cinder/cinder/volume/drivers/remotefs.py:250
2016-02-10 18:58:59.858 DEBUG oslo_concurrency.processutils [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] CMD "mount" returned: 0 in 0.011s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
2016-02-10 18:58:59.859 INFO os_brick.remotefs.remotefs [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] Already mounted: /nfs_storage/cfad14cfb6e7127df26c9db03721a02a
2016-02-10 18:58:59.860 DEBUG cinder.volume.drivers.remotefs [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] Available shares ['192.168.104.11:/storage'] _ensure_shares_mounted /opt/stack/cinder/cinder/volume/drivers/remotefs.py:276
```

图 8-161

执行 stat、du 命令检查 NFS 目录, 如图 8-162 所示。

```
2016-02-10 18:58:59.883 DEBUG oslo_concurrency.processutils [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] Running cmd (subprocess): stat -f -c %s %b %a /nfs_storage/cfad14cfb6e7127df26c9db03721a02a execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-02-10 18:58:59.893 DEBUG oslo_concurrency.processutils [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] CMD "stat -f -c %s %b %a /nfs_storage/cfad14cfb6e7127df26c9db03721a02a" returned: 0 in 0.010s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
2016-02-10 18:58:59.895 DEBUG oslo_concurrency.processutils [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] Running cmd (subprocess): du -sb --apparent-size --exclude *snapshot* /nfs_storage/cfad14cfb6e7127df26c9db03721a02a execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-02-10 18:58:59.906 DEBUG oslo_concurrency.processutils [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] CMD "du -sb --apparent-size --exclude *snapshot* /nfs_storage/cfad14cfb6e7127df26c9db03721a02a" returned: 0 in 0.011s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
```

图 8-162

执行 truncate 创建 volume 文件, 如图 8-163 所示。

```
2016-02-10 18:58:59.907 DEBUG cinder.volume.drivers.nfs [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] Selected 192.168.104.11:/storage as target nfs share. _find_share /opt/stack/cinder/cinder/volume/drivers/nfs.py:216
2016-02-10 18:58:59.908 INFO cinder.volume.drivers.remotefs [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] casted to 192.168.104.11:/storage
2016-02-10 18:58:59.909 DEBUG oslo_concurrency.processutils [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] Running cmd (subprocess): truncate -s 1G /nfs_storage/cfad14cfb6e7127df26c9db03721a02a/volume-7a92a9dd-d1c5-43ef-929a-c00e9a4fd576 execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-02-10 18:58:59.937 DEBUG oslo_concurrency.processutils [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] CMD "truncate -s 1G /nfs_storage/cfad14cfb6e7127df26c9db03721a02a/volume-7a92a9dd-d1c5-43ef-929a-c00e9a4fd576" returned: 0 in 0.028s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
```

图 8-163

设置 volume 文件为可读写，如图 8-164 所示。

```
2016-02-10 18:58:59.953 DEBUG oslo_concurrency.processutils [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] CMD "chmod ugo+rw /nfs-storage/cfad14cfb6e7127df26c9db03721a02a/volume-7a92a9dd-d1c5-43ef-929a-c00e9a4fd576" returned: 0 in 0.014s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
```

图 8-164

create 操作完成，如图 8-165 所示。

```
2016-02-10 18:59:00.031 DEBUG cinder.volume.manager [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] Task 'cinder.volume.flows.manager.create_volume.CreateVolumeFromSpecTask;volume:create' (51f821d7-592a-40ba-9421-3c93d85a6517) transitioned into state 'SUCCESS' from state 'RUNNING' with result '<cinder.db.sqlalchemy.models.Volume object at 0x7fdbfd5f3290>' _task_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:178
2016-02-10 18:59:00.032 DEBUG cinder.volume.manager [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] Task 'cinder.volume.flows.manager.create_volume.CreateVolumeOnFinishTask;volume:create, create.er d' (d2829f53-91a3-47b9-b4d3-69d34b822025) transitioned into state 'RUNNING' from state 'PENDING' _task_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:189
2016-02-10 18:59:00.187 INFO cinder.volume.manager.create_vo lume [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] volume volum e-7a92a9dd-d1c5-43ef-929a-c00e9a4fd576 (7a92a9dd-d1c5-43ef-929a-c0 0e9a4fd576): created successfully
2016-02-10 18:59:00.188 DEBUG cinder.volume.manager [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] Task 'cinder.volume.flows.manager.create_volume.CreateVolumeOnFinishTask;volume:create, create.er d' (d2829f53-91a3-47b9-b4d3-69d34b822025) transitioned into state 'SUCCESS' from state 'RUNNING' with result 'None' _task_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:178
2016-02-10 18:59:00.191 DEBUG cinder.volume.manager [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] Flow 'volume_create_manager' (7b1e7234-4ab8-409b-91d4-4fa848c3a66e) transitioned into state 'SUCCESS' from state 'RUNNING' _flow_receiver /usr/local/lib/python2.7/dist-packages/taskflow/listeners/logging.py:140
2016-02-10 18:59:00.192 INFO cinder.volume.manager [req-783d809a-dea7-485a-bff8-70b50ef5e752 admin] [volume-7a92a9dd-d1c5-43ef-929a-c00e9a4fd576] created volume successfully.
```

图 8-165

Volume 在 NFS 上以文件存在，命名为“volume-<volume\_id>”，如图 8-166 所示。

```
root@devstack-controller:~# ls -l /nfs_storage/cfad14cfb6e7127df26c9db03721a02a/
total 0
-rw-rw-rw- 1 root root 1073741824 Feb 10 18:59 volume-7a92a9dd-d1c5-43ef-929a-c00e9a4fd576
root@devstack-controller:~#
```

图 8-166

GUI volume 列表中可以看到新创建的 volume，如图 8-167 所示。

<input type="checkbox"/>	Name	Description	Size	Status	Type
<input type="checkbox"/>	nfs-vol-1	-	1GB	Available	nfs

图 8-167



### (3) Attach volume

下面我们将前一小节创建的 NFS volume “nfs-vol-1” attach 到 instance “c2” 上。

这里我们重点关注 nova-compute 如何将 “nfs-vol-1” attach 到 “c2”。

通过日志分析, nova-compute 会将存放 volume 文件的 NFS 目录 mount 到本地 /opt/stack/data/nova/mnt 目录下, 然后修改 instance 的 XML 将 volume 文件配置为虚拟磁盘, 日志为 /opt/stack/logs/n-cpu.log

通过 findmnt 和 mkdir 测试和创建 mount point, 如图 8-168 所示。

```
2016-02-10 19:34:18.719 DEBUG oslo_concurrency.processutils [req-a7f093a7-6f24-4ba5-bfa4-253d1fc01c74 admin admin] Running cmd (subprocess): findmnt --target /opt/stack/data/nova/mnt/cfad14cfb6e7127df26c9db03721a02a --source 192.168.104.11:/storage execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-02-10 19:34:18.733 DEBUG oslo_concurrency.processutils [req-a7f093a7-6f24-4ba5-bfa4-253d1fc01c74 admin admin] CMD "findmnt --target /opt/stack/data/nova/mnt/cfad14cfb6e7127df26c9db03721a02a --source 192.168.104.11:/storage" returned: 1 in 0.013s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
2016-02-10 19:34:18.734 DEBUG oslo_concurrency.processutils [req-a7f093a7-6f24-4ba5-bfa4-253d1fc01c74 admin admin] u'findmnt --target /opt/stack/data/nova/mnt/cfad14cfb6e7127df26c9db03721a02a --source 192.168.104.11:/storage' failed. Not Retrying. execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:328
2016-02-10 19:34:18.734 DEBUG oslo_concurrency.processutils [req-a7f093a7-6f24-4ba5-bfa4-253d1fc01c74 admin admin] Running cmd (subprocess): mkdir -p /opt/stack/data/nova/mnt/cfad14cfb6e7127df26c9db03721a02a execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
2016-02-10 19:34:18.747 DEBUG oslo_concurrency.processutils [req-a7f093a7-6f24-4ba5-bfa4-253d1fc01c74 admin admin] CMD "mkdir -p /opt/stack/data/nova/mnt/cfad14cfb6e7127df26c9db03721a02a" returned: 0 in 0.013s execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:280
```

图 8-168

mount NFS 目录, 如图 8-169 所示。

```
2016-02-10 19:34:18.748 DEBUG oslo_concurrency.processutils [req-a7f093a7-6f24-4ba5-bfa4-253d1fc01c74 admin admin] Running cmd (subprocess): mount -t nfs 192.168.104.11:/storage /opt/stack/data/nova/mnt/cfad14cfb6e7127df26c9db03721a02a execute /usr/local/lib/python2.7/dist-packages/oslo_concurrency/processutils.py:250
```

图 8-169

更新 instance 的 XML 配置文件, 将 volume 文件映射给 instance, 如图 8-170 所示。

```
2016-02-10 19:34:18.906 DEBUG nova.virt.libvirt.config [req-a7f093a7-6f24-4ba5-bfa4-253d1fc01c74 admin admin] Generated XML ('<disk type="file" device="disk">\n <driver name="qemu" type="raw" cache="none"/>\n <source file="/opt/stack/data/nova/mnt/cfad14cfb6e7127df26c9db03721a02a/volume-7a92a9dd-d1c5-43ef-929a-c00e9a4fd576"/>\n <target bus="virtio" dev="vdc"/>\n <serial>7a92a9dd-d1c5-43ef-929a-c00e9a4fd576</serial>\n</disk>\n',) to_xml /opt/stack/nova/nova/virt/libvirt/config.py:82
```

图 8-170

我们也可以通过 virsh edit <instance> 查看更新后的 XML, 如图 8-171 所示。



## 第 9 章

# Networking Service ——Neutron

本章我们学习 OpenStack 的 Networking Service——Neutron。

## 9.1 Neutron 概述

传统的网络管理方式很大程度上依赖于管理员手工配置和维护各种网络硬件设备；而云环境下的网络已经变得非常复杂，特别是在多租户场景里，用户随时都可能需要创建、修改和删除网络，网络的连通性和隔离已经太可能通过手工配置来保证了。

如何快速响应业务的需求对网络管理提出了更高的要求。传统的网络管理方式已经很难胜任这项工作，而“软件定义网络（software-defined networking, SDN）”所具有的灵活性和自动化优势使其成为云时代网络管理的主流。

Neutron 的设计目标是实现“网络即服务（Networking as a Service）”。为了达到这一目标，在设计上遵循了基于 SDN 实现网络虚拟化的原则，在实现上充分利用了 Linux 系统上的各种网络相关的技术。

在这一章，我们将讨论 Neutron 的功能和它的各个组件，学习部署和配置 OpenStack 网络的不同方法，会涉及软件和硬件设备多个层面。

### 9.1.1 Neutron 功能

Neutron 为整个 OpenStack 环境提供网络支持，包括二层交换，三层路由，负载均衡，防火墙和 VPN 等。Neutron 提供了一个灵活的框架，通过配置，无论是开源还是商业软件都可以被用来实现这些功能。

#### 1. 二层交换 Switching

Nova 的 Instance 是通过虚拟交换机连接到虚拟二层网络的。

Neutron 支持多种虚拟交换机，包括 Linux 原生的 Linux Bridge 和 Open vSwitch。

Open vSwitch (OVS) 是一个开源的虚拟交换机，它支持标准的管理接口和协议。

利用 Linux Bridge 和 OVS, Neutron 除了可以创建传统的 VLAN 网络，还可以创建基于隧道技术的 Overlay 网络，比如 VxLAN 和 GRE (Linux Bridge 目前只支持 VxLAN)。

在后面章节我们会学习如何使用和配置 Linux Bridge 和 Open vSwitch。

## 2. 三层路由 Routing

Instance 可以配置不同网段的 IP, Neutron 的 router (虚拟路由器) 实现 instance 跨网段通信。router 通过 IP forwarding、iptables 等技术来实现路由和 NAT。

我们将在后面章节讨论如何在 Neutron 中配置 router 来实现 instance 之间，以及与外部网络的通信。

## 3. 负载均衡 Load Balancing

Openstack 在 Grizzly 版本第一次引入了 Load-Balancing-as-a-Service (LBaaS)，提供了将负载分发到多个 instance 的能力。LBaaS 支持多种负载均衡产品和方案，不同地实现以 Plugin 的形式集成到 Neutron，目前默认的 Plugin 是 HAProxy。

我们也将后面章节学习 LBaaS 的使用和配置。

## 4. 防火墙 Firewalling

Neutron 通过下面两种方式保障 instance 和网络的安全性。

- Security Group

通过 iptables 限制进出 instance 的网络包。

- Firewall-as-a-Service

FWaaS，限制进出虚拟路由器的网络包，也是通过 iptables 实现。

我们将在后面章节详细讨论 Security 和 FWaaS。

## 9.1.2 Neutron 网络基本概念

Neutron 管理的网络资源包括 network、subnet 和 port，下面依次介绍。

### 1. network

network 是一个隔离的二层广播域。Neutron 支持多种类型的 network，包括 local、fla、VLAN、VxLAN 和 GRE。

- local

local 网络与其他网络和节点隔离。local 网络中的 instance 只能与位于同一节点上同一网络的 instance 通信，local 网络主要用于单机测试。



- flat

flat 网络是无 vlan tagging 的网络。flat 网络中的 instance 能与位于同一网络的 instance 通信，并且可以跨多个节点。

- vlan

vlan 网络是具有 802.1q tagging 的网络。vlan 是一个二层的广播域，同一 vlan 中的 instance 可以通信，不同 vlan 只能通过 router 通信。vlan 网络可以跨节点，是应用最广泛的网络类型。

- vxlan

vxlan 是基于隧道技术的 overlay 网络。vxlan 网络通过唯一的 segmentation ID (也叫 VNI) 与其他 vxlan 网络区分。vxlan 中数据包会通过 VNI 封装成 UPD 包进行传输。因为二层的包通过封装在三层传输，能够克服 vlan 和物理网络基础设施的限制。

- gre

gre 是与 vxlan 类似的一种 overlay 网络。主要区别在于使用 IP 包而非 UDP 进行封装。不同 network 之间在二层上是隔离的。

以 vlan 网络为例，network A 和 network B 会分配不同的 VLAN ID，这样就保证了 network A 中的广播包不会跑到 network B 中。当然，这里的隔离是指二层上的隔离，借助路由器不同，network 是可能在三层上通信的。

network 必须属于某个 Project (Tenant 租户)，Project 中可以创建多个 network。

network 与 Project 之间是 1 对多关系。

## 2. subnet

subnet 是一个 IPv4 或者 IPv6 地址段。instance 的 IP 从 subnet 中分配。每个 subnet 需要定义 IP 地址的范围和掩码。

network 与 subnet 是 1 对多关系。一个 subnet 只能属于某个 network；一个 network 可以有多个 subnet，这些 subnet 可以是不同的 IP 段，但不能重叠。下面的配置是有效的：

```
network A
subnet A-a: 10.10.1.0/24 {"start": "10.10.1.1", "end": "10.10.1.50"}
subnet A-b: 10.10.2.0/24 {"start": "10.10.2.1", "end": "10.10.2.50"}
```

但下面的配置则无效，因为 subnet 有重叠：

```
network A
subnet A-a: 10.10.1.0/24 {"start": "10.10.1.1", "end": "10.10.1.50"}
subnet A-b: 10.10.1.0/24 {"start": "10.10.1.51", "end": "10.10.1.100"}
```

这里不是判断 IP 是否有重叠，而是 subnet 的 CIDR 重叠（都是 10.10.1.0/24）。

但是，如果 subnet 在不同的 network 中，CIDR 和 IP 都是可以重叠的，比如：

```
network A subnet A-a: 10.10.1.0/24 {"start": "10.10.1.1", "end":  
"10.10.1.50"}  
network B subnet B-a: 10.10.1.0/24 {"start": "10.10.1.1", "end":  
"10.10.1.50"}
```

这里大家不免会疑惑：

如果上面的 IP 地址是可以重叠的，那么就可能存在具有相同 IP 的两个 instance，这样会不会冲突？

简单的回答是：不会！

具体原因是：因为 Neutron 的 router 是通过 Linux network namespace 实现的。network namespace 是一种网络的隔离机制。通过它，每个 router 有自己独立的路由表。

上面的配置有两种结果：

- 如果两个 subnet 是通过同一个 router 路由，根据 router 的配置，只有指定的一个 subnet 可被路由。
- 如果上面的两个 subnet 是通过不同 router 路由，因为 router 的路由表是独立的，所以两个 subnet 都可以被路由。

这里只是先简单做个说明，我们会在后面三层路由的章节详细分析这种场景。

### 3. port

port 可以看作虚拟交换机上的一个端口。port 上定义了 MAC 地址和 IP 地址，当 instance 的虚拟网卡 VIF (Virtual Interface) 绑定到 port 时，port 会将 MAC 和 IP 分配给 VIF。

port 与 subnet 是 1 对多关系。一个 port 必须属于某个 subnet；一个 subnet 可以有多个 port。

### 4. 小节

下面总结了 Project、Network、Subnet、Port 和 VIF 之间关系。

```
Project 1 : m Network 1 : m Subnet 1 : m Port 1 : 1 VIF m : 1  
Instance
```

## 9.2 Neutron 架构

与 OpenStack 的其他服务的设计思路一样，Neutron 也是采用分布式架构，由多个组件（子服务）共同对外提供网络服务。Neutron 架构如图 9-1 所示。

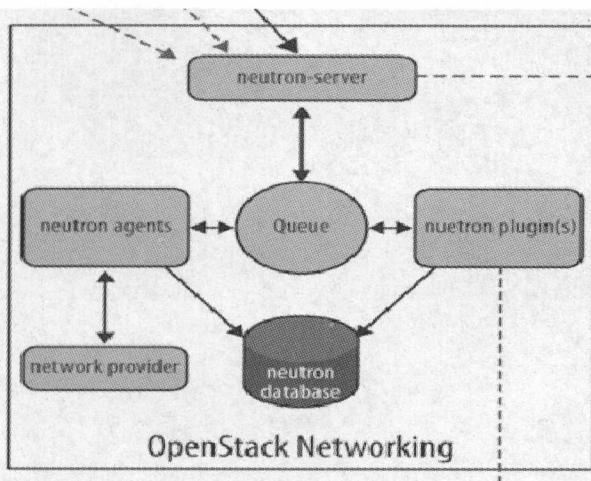


图 9-1

Neutron 由如下组件构成:

- Neutron Server

对外提供 OpenStack 网络 API, 接收请求, 并调用 Plugin 处理请求。

- Plugin

处理 Neutron Server 发来的请求, 维护 OpenStack 逻辑网络的状态, 并调用 Agent 处理请求。

- Agent

处理 Plugin 的请求, 负责在 network provider 上真正实现各种网络功能。

- network provider

提供网络服务的虚拟或物理网络设备, 例如 Linux Bridge, Open vSwitch 或者其他支持 Neutron 的物理交换机。

- Queue

Neutron Server、Plugin 和 Agent 之间通过 Messaging Queue 通信和调用。

- Database

Database 用来存放 OpenStack 的网络状态信息, 包括 Network、Subnet、Port、Router 等。如图 9-2 所示。

```
root@devstack-controller:~# su - stack
stack@devstack-controller:~$ mysql
welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6652
Server version: 5.5.46-0ubuntu0.14.04.2 (Ubuntu)

Copyright (c) 2000, 2015, Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use neutron
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+-----+
Tables_in_neutron
+-----+
address_scopes
agents
alembic_version
alembic_version_fwaas
alembic_version_lbaas
allowedaddresspairs
arista_provisioned_nets
arista_provisioned_tenants
arista_provisioned_vms
```

图 9-2

Neutron 架构非常灵活，层次较多，其目的是：

- 为了支持各种现有或者将来会出现的优秀网络技术。
- 支持分布式部署，获得足够的扩展性。

通常鱼和熊掌不能兼得，虽然获得了这些优势，但 Neutron 也变得更复杂，更不容易理解。

后面我们会详细讨论 Neutron 的各个组件，但在这之前，非常有必要先通过一个例子了解这些组件各自的职责以及如何协同工作。

以创建一个 VLAN100 的 network 为例，假设 network provider 是 Linux Bridge，流程如下：

- Neutron Server 接收到创建 network 的请求，通过 Message Queue (RabbitMQ) 通知已注册的 Linux Bridge Plugin。
- Plugin 将要创建的 network 的信息（例如名称、VLAN ID 等）保存到数据库中，并通过 Message Queue 通知运行在各节点上的 Agent。
- Agent 收到消息后会在节点上的物理网卡（比如 eth2）上创建 VLAN 设备（比如 eth2.100），并创建 bridge（比如 brqXXX）桥接 VLAN 设备。

关于 linux bridge 如何实现 VLAN 大家可以参考本教程“预备知识 → 网络虚拟化”的相关章节。

这里进行几点说明：

plugin 解决的是 What 的问题，即网络要配置成什么样子？而至于如何配置 How 的工作则交由 agent 完成。

plugin、agent 和 network provider 是配套使用的，比如上例中 network provider 是 Linux Bridge，那么就得使用 Linux Bridge 的 plugin 和 agent。如果 network provider 换成了 OVS 或者物



理交换机, plugin 和 agent 也得替换。

plugin 的一个主要的职责是在数据库中维护 Neutron 网络的状态信息, 这就造成一个问题: 所有 network provider 的 plugin 都要编写一套非常类似的数据库访问代码。为了解决这个问题, Neutron 在 Havana 版本实现了一个 ML2 (Modular Layer 2) plugin, 对 plugin 的功能进行抽象和封装。有了 ML2 plugin, 各种 network provider 无须开发自己的 plugin, 只需要针对 ML2 开发相应的 driver 就可以了, 工作量和难度都大大减少。ML2 会在后面详细讨论。

plugin 按照功能分为两类: core plugin 和 service plugin。core plugin 维护 Neutron 的 network、subnet 和 port 相关资源的信息, 与 core plugin 对应的 agent 包括 Linux Bridge、OVS 等。service plugin 提供 routing、firewall、load balance 等服务, 也有相应的 agent。后面也会分别详细讨论。

## 9.2.1 物理部署方案

本节讨论 Neutron 的物理部署方案: 不同节点部署不同的 Neutron 服务组件。

### 1. 方案1: 控制节点 + 计算节点

在这个部署方案中, OpenStack 由控制节点和计算节点组成。

#### ● 控制节点

部署的服务包括: neutron server、core plugin 的 agent 和 service plugin 的 agent。

#### ● 计算节点

部署 core plugin 的 agent, 负责提供二层网络功能。

这里有两点需要说明:

- (1) core plugin 和 service plugin 已经集成到 neutron server, 不需要运行独立的 plugin 服务。
- (2) 控制节点和计算节点都需要部署 core plugin 的 agent, 因为通过该 agent 控制节点与计算节点才能建立二层连接。
- (3) 可以部署多个控制节点和计算节点。

### 2. 方案2: 控制节点 + 网络节点 + 计算节点

在这个部署方案中, OpenStack 由控制节点、网络节点和计算节点组成。

#### ● 控制节点

部署 neutron server 服务。

#### ● 网络节点

部署的服务包括: core plugin 的 agent 和 service plugin 的 agent。

- 计算节点

部署 core plugin 的 agent，负责提供二层网络功能。

这个方案的要点是将所有的 agent 从控制节点分离出来，部署到独立的网络节点上。

- 控制节点只负责通过 neutron server 响应 API 请求。
- 由独立的网络节点实现数据的交换、路由以及 load balance 等高级网络服务。
- 可以通过增加网络节点承担更大的负载。
- 可以部署多个控制节点、网络节点和计算节点。

该方案特别适合规模较大的 OpenStack 环境。

## 9.2.2 Neutron Server

本节将详细讨论 Neutron Server 服务。Neutron Server 的分层结构如图 9-3 所示。

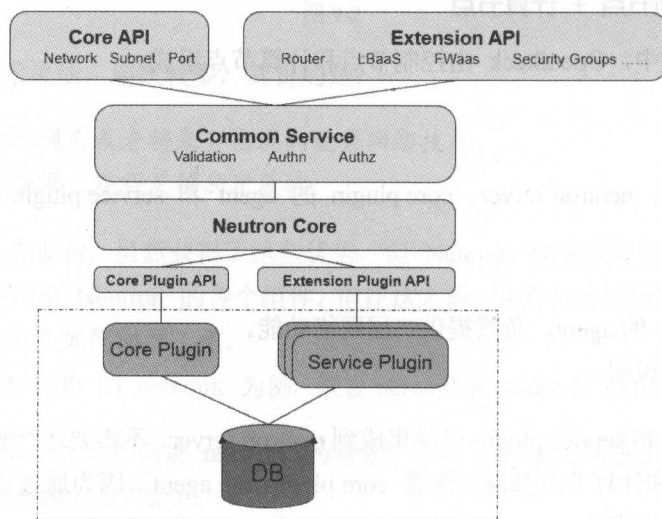


图 9-3

Neutron Server 的分层结构，自上而下依次为：

- Core API

对外提供管理 network、subnet 和 port 的 RESTful API。

- Extension API

对外提供管理 router、load balance、firewall 等资源的 RESTful API。

- Common Service

认证和校验 API 请求。

- Neutron Core

Neutron server 的核心处理程序，通过调用相应的 Plugin 处理请求。

- Core Plugin API

定义了 Core Plgin 的抽象功能集合，Neutron Core 通过该 API 调用相应的 Core Plgin。

- Extension Plugin API

定义了 Service Plgin 的抽象功能集合，Neutron Core 通过该 API 调用相应的 Service Plgin。

- Core Plugin

实现了 Core Plugin API，在数据库中维护 network、subnet 和 port 的状态，并负责调用相应的 agent 在 network provider 上执行相关操作，比如创建 network。

- Service Plugin

实现了 Extension Plugin API，在数据库中维护 router、load balance、security group 等资源的状态，并负责调用相应的 agent 在 network provider 上执行相关操作，比如创建 router。

归纳起来，Neutron Server 包括两部分功能：提供 API 服务与运行 Plugin，即 Neutron Server = API + Plugins，如图 9-4 所示。

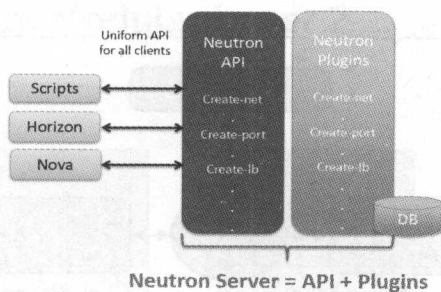


图 9-4

### 9.2.3 Neutron 如何支持各种 network provider

先讨论一个简单的场景：在 Neutron 中使用 Linux Bridge 这一种 network provider。

根据我们上一节讨论的 Neutron Server 的分层模型，我们需要实现两个东西：linux bridge core plugin 和 linux bridge agent。

- linux bridge core plugin.

- 与 neutron server 一起运行。
- 实现了 core plugin API。
- 负责维护数据库信息。

- 通知 linux bridge agent 实现具体的网络功能。
- linux bridge agent
  - 在计算节点和网络节点（或控制节点）上运行。
  - 接收来自 plugin 的请求。
  - 通过配置本节点上的 linux bridge 实现 neutron 网络功能，如图 9-5 所示。

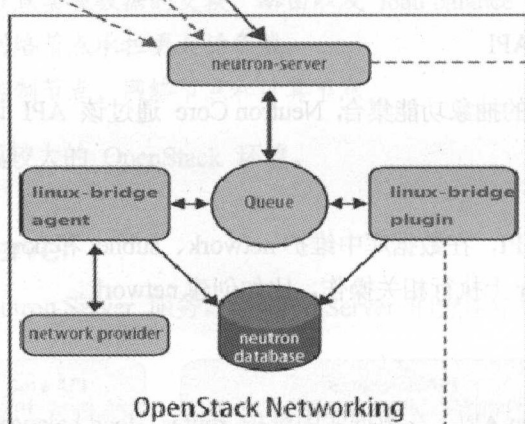


图 9-5

同样的道理，如果要支持 open vswitch，只需要实现 open vswitch plugin 和 open vswitch agent，如图 9-6 所示。

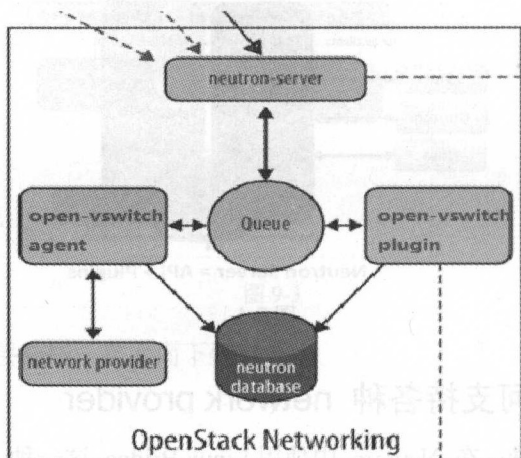


图 9-6

由此可见：Neutron 可以通过开发不同的 plugin 和 agent 支持不同的网络技术。这是一种相当开放的架构。

不过随着支持的 network provider 数量的增加，开发人员发现了两个突出的问题：

- 只能在 OpenStack 中使用一种 core plugin，多种 network provider 无法共存。



- 不同 plugin 之间存在大量重复代码，开发新的 plugin 工作量大。

下一节将深入讨论这两个问题的成因以及解决方案。

## 9.2.4 ML2 Core Plugin

Moduler Layer 2 (ML2) 是 Neutron 在 Havana 版本实现的一个新的 core plugin，用于替代原有的 linux bridge plugin 和 open vswitch plugin。

### 1. 传统 core plugin 的问题

之所以要开发 ML2，主要是因为传统 core plugin 存在两个突出的问题。

#### (1) 问题 1：无法同时使用多种 network provider

Core plugin 负责管理和维护 Neutron 的 network、subnet 和 port 的状态信息，这些信息是全局的，只需要也只能由一个 core plugin 管理。

只使用一个 core plugin 本身没有问题。但问题在于传统的 core plugin 与 core plugin agent 是一一对应的。也就是说，如果选择了 linux bridge plugin，那么 linux bridge agent 将是唯一选择，就必须在 OpenStack 的所有节点上使用 Linux Bridge 作为虚拟交换机（即 network provider）。

同样的，如果选择 open vswitch plugin，所有节点上只能使用 open vswitch，而不能使用其他的 network provider，如图 9-7 所示。

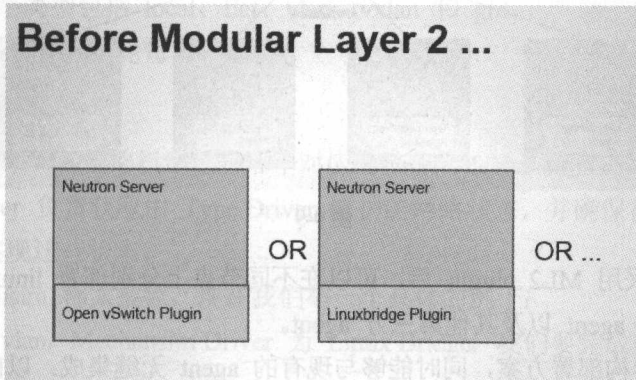


图 9-7

#### (2) 问题 2：开发新的 core plugin 工作量大

所有传统的 core plugin 都需要编写大量重复和类似的数据库访问的代码，大大增加了 plugin 开发和维护的工作量，如图 9-8 所示。

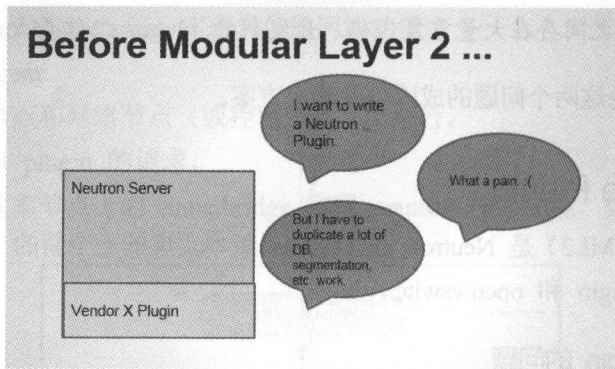


图 9-8

## 2. ML2 能解决传统 core plugin 的问题

ML2 作为新一代的 core plugin，提供了一个框架，允许在 OpenStack 网络中同时使用多种 Layer 2 网络技术，不同的节点可以使用不同的网络实现机制，如图 9-9 所示。

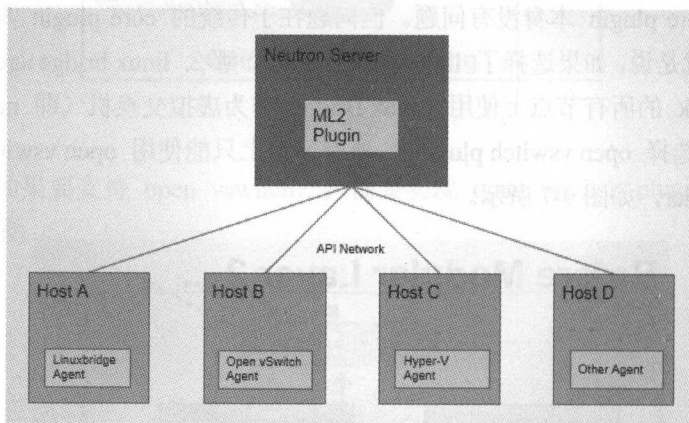


图 9-9

如图 9-9 所示，采用 ML2 plugin 后，可以在不同节点上分别部署 linux bridge agent、open vswitch agent、hyper-v agent 以及其他第三方 agent。

ML2 不但支持异构部署方案，同时能够与现有的 agent 无缝集成：以前用的 agent 不需要变，只需要将 Neutron Server 上的传统 core plugin 替换为 ML2。

有了 ML2，要支持新的 network provider 就变得简单多了：无须从头开发 core plugin，只需要开发相应的 mechanism driver，大大减少了要编写和维护的代码。

## 3. ML2 架构

ML2 对二层网络进行抽象和建模，引入了 type driver 和 mechansim driver。

这两类 driver 解耦了 Neutron 所支持的网络类型（type）与访问这些网络类型的机制（mechanism），其结果就是使得 ML2 具有非常好的弹性，易于扩展，能够灵活支持多种 type 和 mechanism，如图 9-10 所示。

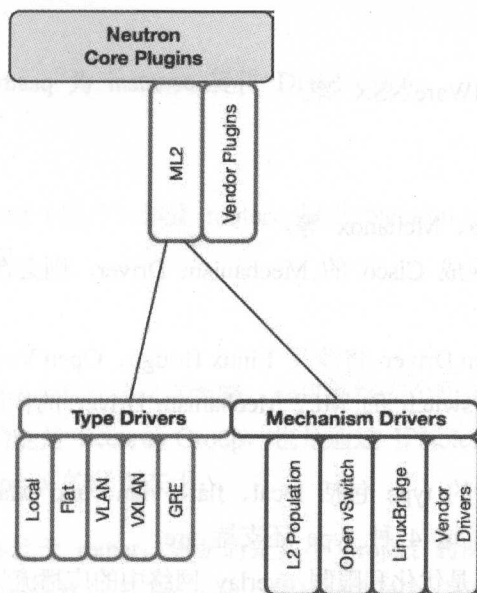


图 9-10

### (1) Type Driver

Neutron 支持的每一种网络类型都有一个对应的 ML2 Type Driver。

Type Driver 负责维护网络类型的状态，执行验证、创建网络等。

ML2 支持的网络类型包括 local、flat、vlan、vxlan 和 gre。

我们将在后面章节详细讨论每种 type。

### (2) Mechanism Driver

Neutron 支持的每一种网络机制都有一个对应的 ML2 Mechanism Driver。

Mechanism Driver 负责获取由 Type Driver 维护的网络状态，并确保在相应的网络设备（物理或虚拟）上正确实现这些状态。

type 和 mechanism 都太抽象，现在我们举一个具体的例子：

Type Driver 为 vlan，Mechanism Driver 为 Linux Bridge，我们要完成的操作是创建 network vlan100，那么：

- vlan type driver 会确保将 vlan100 的信息保存到 Neutron 数据库中，包括 network 的名称，vlan ID 等。
- linux bridge mechanism driver 会确保各节点上的 linux bridge agent 在物理网卡上创建 ID 为 100 的 vlan 设备和 bridge 设备，并将两者进行桥接。

Mechanism Driver 有三种类型：

- Agent-based

包括 Linux Bridge、Open Vswitch 等。

- Controller-based

包括 OpenDaylight、VMWare NSX 等。

- 基于物理交换机

包括 Cisco Nexus、Arista、Mellanox 等。

比如前面那个例子如果换成 Cisco 的 Mechanism Driver，则会在 Cisco 物理交换机的指定 trunk 端口上添加 vlan100。

本教程讨论的 Mechanism Driver 将涉及 Linux Bridge、Open Vswitch 和 L2 population。

Linux Bridge 和 Open Vswitch 的 ML2 Mechanism Driver 的作用是配置各节点上的虚拟交换机。

Linux Bridge Driver 支持的 type 包括 local、flat、vlan、and vxlan。

Open Vswitch Driver 除了这 4 种 type 还支持 gre。

L2 population driver 作用是优化和限制 overlay 网络中的广播流量。

vxlan 和 gre 都属于 overlay 网络。

ML2 core plugin 已经成为 OpenStack Neutron 的首选 plugin，本教程后面会讨论如何在实验环境中配置 ML2 的各种 type 和 mechansim。

### 9.2.5 Service Plugin / Agent

Neutron core plugin 及其 agent 负责将 instance 连接到 OpenStack layer 2 虚拟网络，所提供的资源包括 network、subnet 和 port。

Service Plugin 及其 agent 则提供更丰富的扩展功能，包括路由、load balance、firewall 等，如图 9-11 所示。

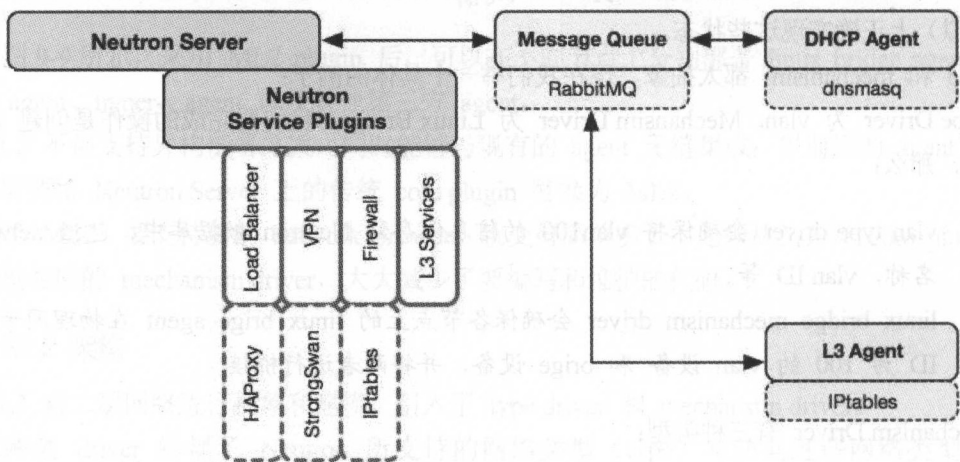


图 9-11



- DHCP

DHCP agent 通过 dnsmasq 为 instance 提供 DHCP 服务。

- Routing

L3 Agent 可以为 project（租户）创建 router，提供 Neutron subnet 之间的路由服务。路由功能默认通过 IPtables 实现。

- Firewall

L3 Agent 可以在 router 上配置防火墙策略，提供网络安全防护。

另一个与安全相关的功能是 Security Group，也是通过 IPtables 实现。

Firewall 与 Security Group 的区别在于：

- Firewall 安全策略位于 router，保护的是某个 project 的所有 network。
- Security Group 安全策略位于 instance，保护的是单个 instance。

Firewall 与 Security Group 后面会详细分析。

- Load Balance

Neutron 默认通过 HAProxy 为 project 中的多个 instance 提供 load balance 服务。

后面的章节会结合 Linux Bridge 和 Open Vswitch 详细讨论每一种 service。

## 9.2.6 小结

前面我们详细讨论的 Neutron 架构，现在用一张图做个总结（见图 9-12）：

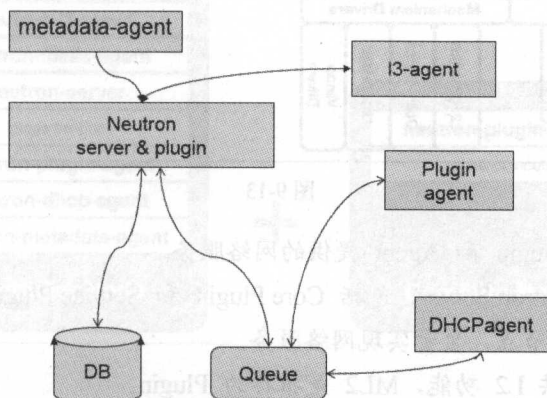


图 9-12

与 OpenStack 其他服务一样，Neutron 采用的是分布式架构，包括 Neutron Server、各种 plugin/agent、database 和 message queue。

- Neutron server 接收 api 请求。
- plugin/agent 实现请求。
- database 保存 neutron 网络状态。
- message queue 实现组件之间通信。

metadata-agent 之前没有讲到，这里做个补充：

instance 在启动时需要访问 nova-metadata-api 服务获取 metadata 和 userdata，这些 data 是该 instance 的定制化信息，比如 hostname、ip、public key 等。但 instance 启动时并没有 ip，如何能够通过网络访问到 nova-metadata-api 服务呢？

答案就是 neutron-metadata-agent。该 agent 让 instance 能够通过 dhcp-agent 或者 l3-agent 与 nova-metadata-api 通信。

如果我们将 Neutron 架构展开，则会得到图 9-13。

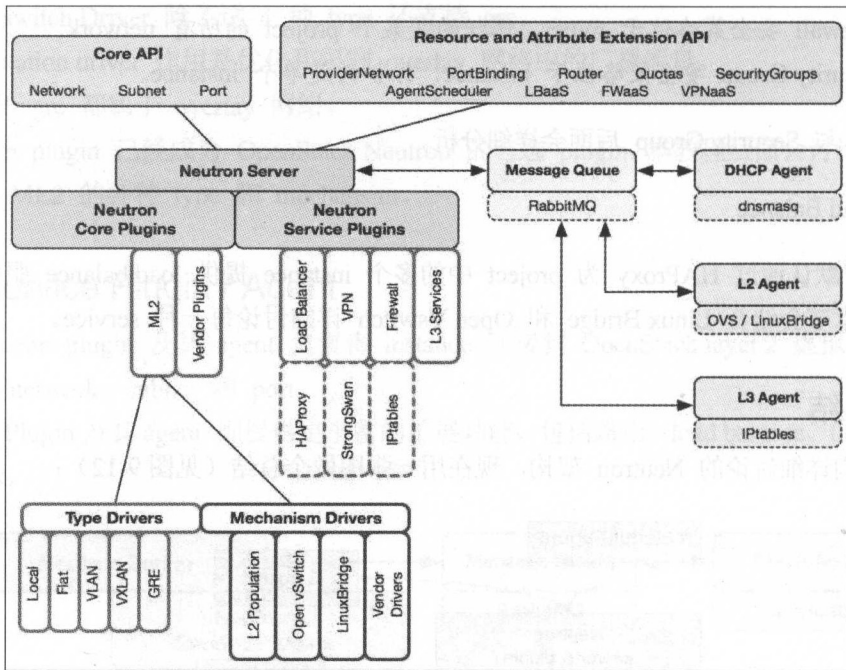


图 9-13

- Neutron 通过 Plugin 和 Agent 提供的网络服务。
- plugin 位于 Neutron Server，包括 Core Plugin 和 Service Plugin。
- Agent 位于各个节点，负责实现网络服务。
- Core Plugin 提供 L2 功能，ML2 是推荐的 Plugin。
- 使用最广泛的 L2 Agent 是 Linux Bridge 和 Open Vswitch。
- Service Plugin 和 Agent 提供扩展功能，包括 DHCP、routing、load balance、firewall、vpn 等。

## 9.3 为 Neutron 准备物理基础设施

前面讨论了 Neutron 的架构和基础知识，接下来就要通过实验深入学习和实践了。

第一步就是准备实验用的物理环境，需要考虑如下几个问题：

- 需要几个节点？
- 如何分配节点的角色？
- 节点上分别部署哪些服务？
- 节点上配置几个网卡？
- 物理网络如何连接？

### 9.3.1 1 控制节点 + 1 计算节点的部署方案

我们的目的是通过实验学习 Neutron 的各种特性。

为了达到这个目的，实验环境应尽量贴近典型的部署方案；但同时，由于是个人学习使用，受物理条件的限制需要尽量利用有限的资源，所以我们采用图 9-14 的部署方案：

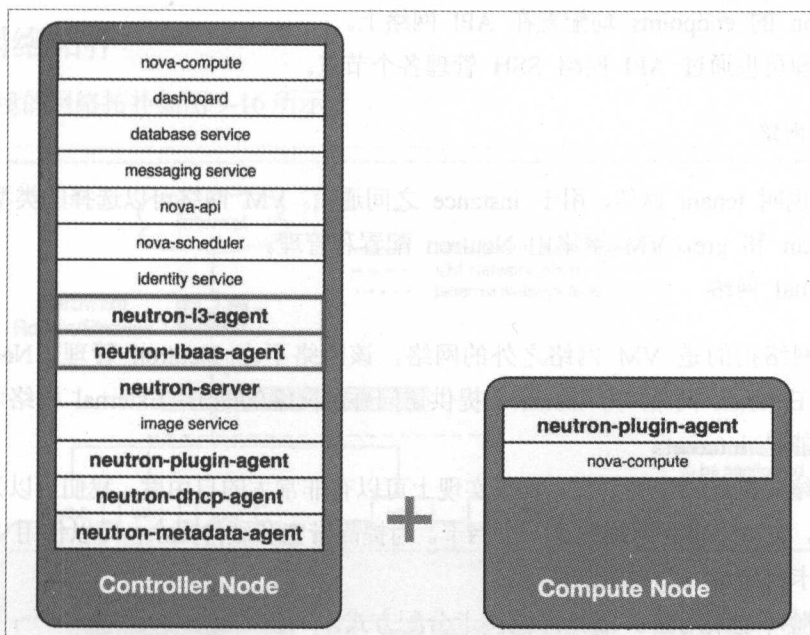


图 9-14

Q: 需要几个节点？

A: 2 节点 = 1 控制节点 + 1 计算节点。

Q: 如何分配节点的角色？

A: 控制节点合并了网络节点的功能，同时也是一个计算节点。

Q: 节点上分别部署哪些服务?

A: 如图 9-14 所示。

### 9.3.2 配置多个网卡区分不同类型的网络数据

逻辑上, OpenStack 至少包含下面几类网络流量:

```
Management
API
VM
External
```

- Management 网络

用于节点之间 message queue 内部通信以及访问 database 服务, 所有的节点都需要连接到 management 网络。

- API 网络

OpenStack 各组件通过该网络向用户暴露 API 服务。Keystone、Nova、Neutron、Glance、Cinder、Horizon 的 endpoints 均配置在 API 网络上。

通常, 管理员也通过 API 网络 SSH 管理各个节点。

- VM 网络

VM 网络也叫 tenant 网络, 用于 instance 之间通信。VM 网络可以选择的类型包括 local、flat、vlan、vxlan 和 gre。VM 网络由 Neutron 配置和管理。

- External 网络

External 网络指的是 VM 网络之外的网络, 该网络不由 Neutron 管理。Neutron 可以将 router 连接到 External 网络, 为 instance 提供访问外部网络的能力。External 网络可能是企业的 intranet, 也可能是 internet。

这几类网络只是逻辑上的划分, 物理实现上可以有非常大的自由度。我们可以为每种网络分配单独的网卡, 也可以多种网络共享一个网卡。为提高带宽和硬件冗余, 可以使用 bonding 技术将多个物理网卡绑定成一个逻辑的网卡。

我们的实验环境采用图 9-15 所示的网卡分配方式。



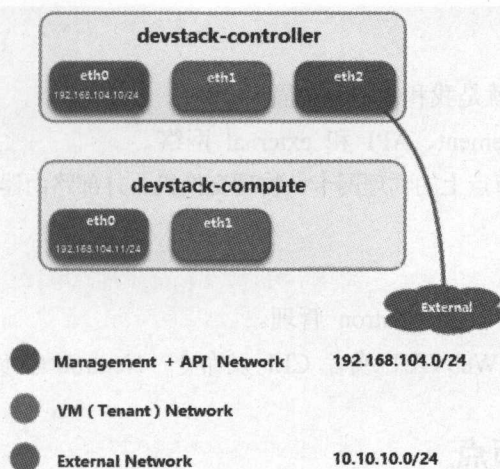


图 9-15

- 控制节点三个网卡 (eth0、eth1、eth2)，计算节点两网卡 (eth0、eth1)。
- 合并 Management 和 API 网络，使用 eth0，IP 段为 192.168.104.0/24。
- VM 网络使用 eth1。
- 控制节点的 eth2 与 External 网络连接，IP 段为 10.10.10.0/24。

### 9.3.3 网络拓扑

实验环境的网络拓扑如图 9-16 所示。

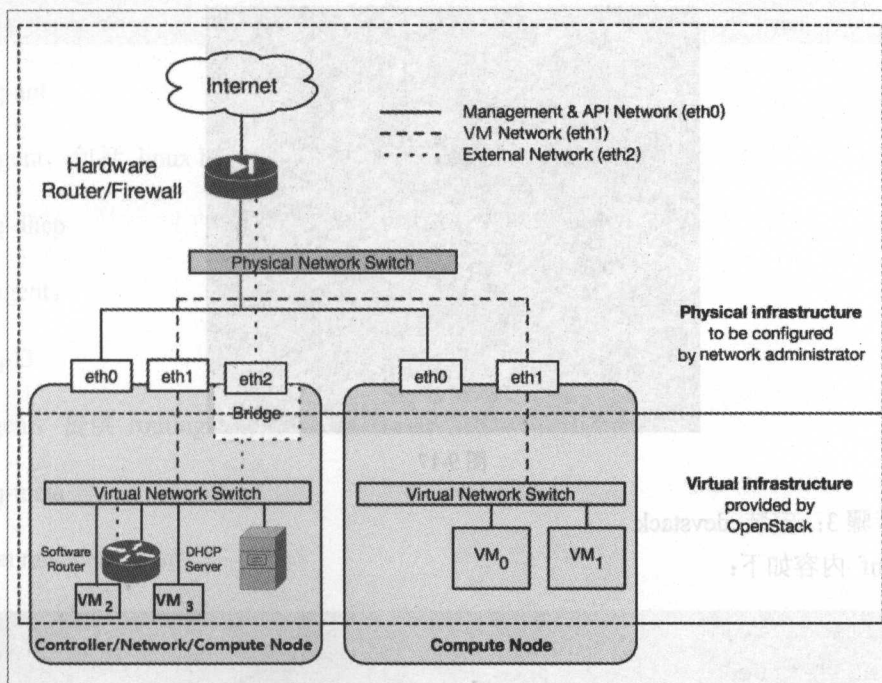


图 9-16

分割线上方的网络:

- (1) 由网络管理员（就是我和你啦）配置。
- (2) 主要涉及 Management、API 和 external 网络。
- (3) 配置的内容包括节点上的物理网卡、物理交换机、外部路由器，防火墙以及物理连线等。

分割线下方的网络:

- (1) 主要是 VM 网络，由 Neutron 管理。
- (2) 我们只需要通过 Web GUI 或者 CLI 发命令，Neutron 会负责实现。

### 9.3.4 安装和配置节点

按照前面的规划安装和配置控制节点和计算节点。

#### 1. 安装配置控制节点 devstack-controller

- (1) 步骤 1: 安装 Ubuntu 14.04

此处省略。

- (2) 步骤 2: 配置网卡

编辑 /etc/network/interfaces。

给 eth0 配置 IP 192.168.104.10，并激活 eth1 和 eth2，如图 9-17 所示。

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
address 192.168.104.10
netmask 255.255.255.0
gateway 192.168.104.254

auto eth1
iface eth1 inet manual

auto eth2
iface eth2 inet manual
```

图 9-17

- (3) 步骤 3: 安装 devstack

local.conf 内容如下:

```
[[local|localrc]]
MULTI_HOST=true
HOST_IP=192.168.104.10 # management & api network
```

```

LOGFILE=/opt/stack/logs/stack.sh.log
# Credentials
ADMIN_PASSWORD=admin
MYSQL_PASSWORD=secret
RABBIT_PASSWORD=secret
SERVICE_PASSWORD=secret
SERVICE_TOKEN=abcdefghijklmnopqrstuvwxyz
# enable neutron-ml2-vlan
disable_service n-net
enable_service
q-svc,q-agt,q-dhcp,q-l3,q-meta,neutron,q-lbaas,q-fwaas,q-vpn
Q_AGENT=linuxbridge
ENABLE_TENANT_VLANS=True
TENANT_VLAN_RANGE=3001:4000
PHYSICAL_NETWORK=default
LOG_COLOR=False
LOGDIR=$DEST/logs
SCREEN_LOGDIR=$LOGDIR/screen

```

值得注意的是我们通过 `enable_service` 指定安装了若干服务。

```

enable_service
q-svc,q-agt,q-dhcp,q-l3,q-meta,neutron,q-lbaas,q-fwaas,q-vpn

```

- q-agt

core agent, 包括 linux bridge agent 和 open vswitch agent。

- q-dhcp

dhcp agent。

- q-l3

L3 Agent, 提供 routing 服务。

- q-meta

neutron metadata agent。

- q-lbaas

load balance agent。

- q-fwass

firewall 服务。

- q-vpn

vpn agent, 提供 VPN as a Service。

另外, 为了加快安装速度, 还可以加上下面的配置, 使用国内的 devstack 镜像站点。

```
# use TryStack git mirror
GIT_BASE=http://git.trystack.cn
NOVNC_REPO=http://git.trystack.cn/kanaka/novnc.git
SPICE_REPO=http://git.trystack.cn/git/spice/spice-html5.git
```

以 stack 用户身份执行 ./stack 安装 devstack。

## 2. 安装配置计算节点 devstack-compute1

### (1) 步骤 1: 安装 Ubuntu 14.04

此步骤省略。

### (2) 步骤 2: 配置网卡

编辑 /etc/network/interfaces。

给 eth0 配置 IP 192.168.104.11, 并激活 eth1, 如图 9-18 所示。

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
address 192.168.104.11
netmask 255.255.255.0
gateway 192.168.104.254

auto eth1
iface eth1 inet manual
```

图 9-18

### (3) 步骤 3: 安装 devstack

local.conf 内容如下:

```
[[local|localrc]]
MULTI_HOST=true
HOST_IP=192.168.104.11 # management & api network
# Credentials
```



```

ADMIN_PASSWORD=admin
MYSQL_PASSWORD=secret
RABBIT_PASSWORD=secret
SERVICE_PASSWORD=secret
SERVICE_TOKEN=abcdefghijklmnopqrstuvwxyz
# Service information
SERVICE_HOST=192.168.104.10
MYSQL_HOST=$SERVICE_HOST
RABBIT_HOST=$SERVICE_HOST
GLANCE_HOSTPORT=$SERVICE_HOST:9292
Q_HOST=$SERVICE_HOST
KEYSTONE_AUTH_HOST=$SERVICE_HOST
KEYSTONE_SERVICE_HOST=$SERVICE_HOST
CEILOMETER_BACKEND=mongodb
DATABASE_TYPE=mysql
ENABLED_SERVICES=n-cpu,q-agt,neutron
Q_AGENT=linuxbridge
ENABLE_TENANT_VLANS=True
TENANT_VLAN_RANGE=3001:4000
PHYSICAL_NETWORK=default
# vnc config
NOVA_VNC_ENABLED=True
NOVNC_PROXY_URL="http://$SERVICE_HOST:6080/vnc_auto.html"
VNC_SERVER_LISTEN=$HOST_IP
VNC_SERVER_PROXYCLIENT_ADDRESS=$VNC_SERVER_LISTEN
LOG_COLOR=False
LOGDIR=$DEST/logs
SCREEN_LOGDIR=$LOGDIR/screen

```

计算节点只需要安装 nova-compute 和 neutron core agent。

```
ENABLED_SERVICES=n-cpu,q-agt,neutron
```

以 stack 用户身份执行 ./stack 安装 devstack。

至此，我们已经完成了 Neutron 实验环境的搭建工作，后面将深入讨论如何用 Linux Bridge 和 Open Vswitch 实现 Neutron 网络。

## 9.4 Linux Bridge 实现 Neutron 网络

Neutron ML2 plugin 默认使用的 mechanism driver 是 Open vSwitch 而不是 Linux Bridge。那是否还有研究 Linux Bridge 的必要呢？

我的答案是：很有必要！

原因如下：

Linux Bridge 技术非常成熟，而且高效，所以业界很多 OpenStack 方案采用的是 Linux Bridge，比如 Rackspace 的 private cloud。

Open vSwitch 实现的 Neutron 虚拟网络较为复杂，不易理解；而 Linux Bridge 方案更直观。先理解 Linux Bridge 方案后再学习 Open vSwitch 方案会更容易。并且可以通过两种方案的对比更加深入地理解 Neutron 网络。

在深入学习之前，我们先复习一下 Linux Bridge 实现虚拟交换机的基本原理，如图 9-19 所示。

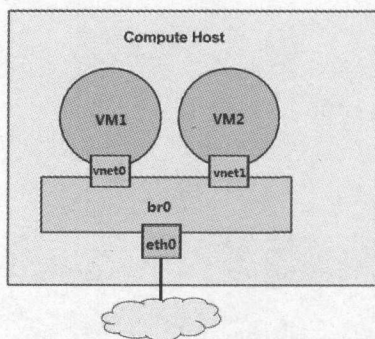


图 9-19

图 9-19 中，br0 是 Linux Bridge，br0 充当虚拟交换机的作用，负责将物理网卡 eth0 和虚拟网卡 tap 设备 vnet0/vent1 连接到同一个二层网络，实现虚拟机 VM1 和 VM2，以及虚拟机与外网之间的通信。

关于 Linux Bridge 更详细的内容请参考“预备知识→网络虚拟化”相关章节。

### 9.4.1 配置 linux-bridge mechanism driver

要在 Neutron 中使用 Linux Bridge，首先需要配置 linux-bridge mechanism driver。

Neutron 默认使用 ML2 作为 core plugin，其配置位于 /etc/neutron/neutron.conf，如图 9-20 所示。

```

[DEFAULT]
api_workers = 2
notify_nova_on_port_data_changes = True
notify_nova_on_port_status_changes = True
auth_strategy = keystone
allow_overlapping_ips = True
debug = True
verbose = True
service_plugins = neutron.services.l3_router.l3_router.plugin.VPNDriverPlugin,neutron.fwaas.services.firewall.f
core_plugin = neutron.plugins.ml2.plugin.ML2Plugin
rpc_backend = rabbit
logging_context_format_string = %(asctime)s.%(msecs)03d
bind_host = 0.0.0.0
use_syslog = False

```

图 9-20

控制节点和计算节点都需要在各自的 `neutron.conf` 中配置 `core_plugin` 选项。

然后需要让 ML2 使用 `linux-bridge mechanism driver`。

ML2 的配置文件位于 `/etc/neutron/plugins/ml2/ml2_conf.ini`，如图 9-21 所示。

```

tenant_network_types = vxlan
extension_drivers = port_security
type_drivers = local,flat,vlan,gre,vxlan
mechanism_drivers = linuxbridge

```

图 9-21

`mechanism_drivers` 选项指明当前节点可以使用的 `mechanism driver`，这里可以指定多种 `driver`，ML2 会负责加载。

上面的配置指明我们只使用 `linux-bridge driver`。

控制节点和计算节点都需要在各自的 `ml2_conf.ini` 中配置 `mechanism_drivers` 选项。

Neutron 服务正常启动后，所有节点上都会运行 `neutron-linuxbridge-agent`，如图 9-22 所示。

```

root@devstack-controller:~# ps -elf|grep linuxbridge
4 S.root      5520  5350  1   80   0 - 32839 ep_pol Mar07 pts/10   01:17:24 /usr/
bin/python /usr/local/bin/neutron-linuxbridge-agent --config-file /etc/neutron/n

```

图 9-22

## 9.4.3

### 9.4.2 初始网络状态

我们首先考察实验环境最初始的网络状态。随着学习的深入，我们会对网络不断进行新的配置，大家也将看到网络一步一步发生的变化。

在我们的实验环境中，当前节点上只存在物理网卡设备 `ethX`，还没有 `bridge` 和 `tap`，状态如下：

- 控制节点及配置，如图 9-23 和图 9-24 所示。



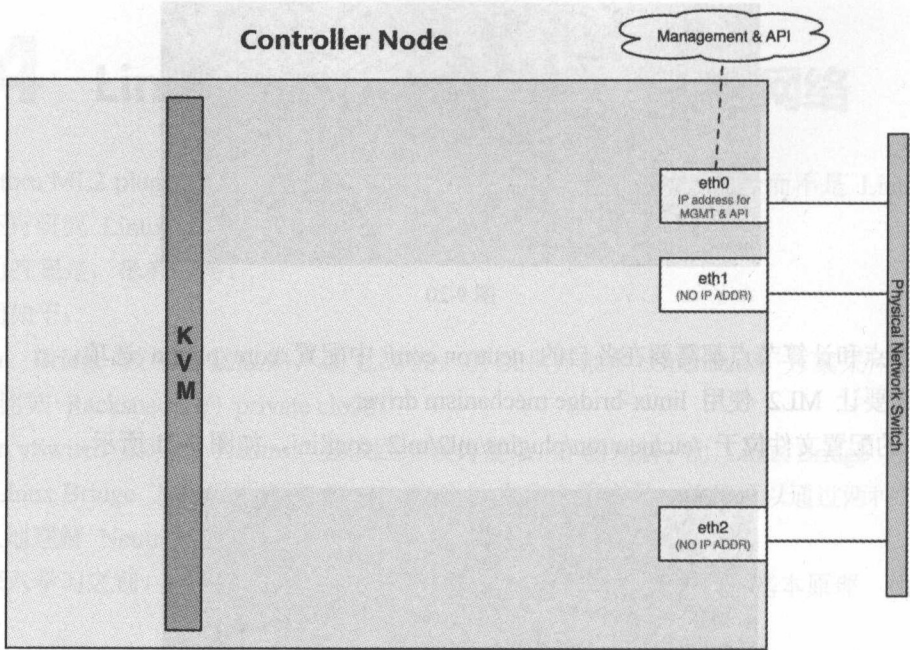


图 9-23

```
root@devstack-controller:~# ifconfig
eth0      Link encap:Ethernet  Hwaddr 00:50:56:b0:ee:49
          inet addr:192.168.104.10  Bcast:192.168.104.255  Mask
          inet6 addr: fe80::250:56ff:feb0:ee49/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:42 errors:0 dropped:0 overruns:0 frame:0
          TX packets:51 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4771 (4.7 KB)  TX bytes:7254 (7.2 KB)

eth1      Link encap:Ethernet  Hwaddr 00:50:56:b0:b5:ff
          inet6 addr: fe80::250:56ff:feb0:b5ff/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:120 (120.0 B)  TX bytes:728 (728.0 B)

eth2      Link encap:Ethernet  Hwaddr 00:50:56:b0:b5:2b
          inet6 addr: fe80::250:56ff:feb0:b52b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:120 (120.0 B)  TX bytes:728 (728.0 B)
```

图 9-24

- 计算节点及配置如图 9-25 和图 9-26 所示。



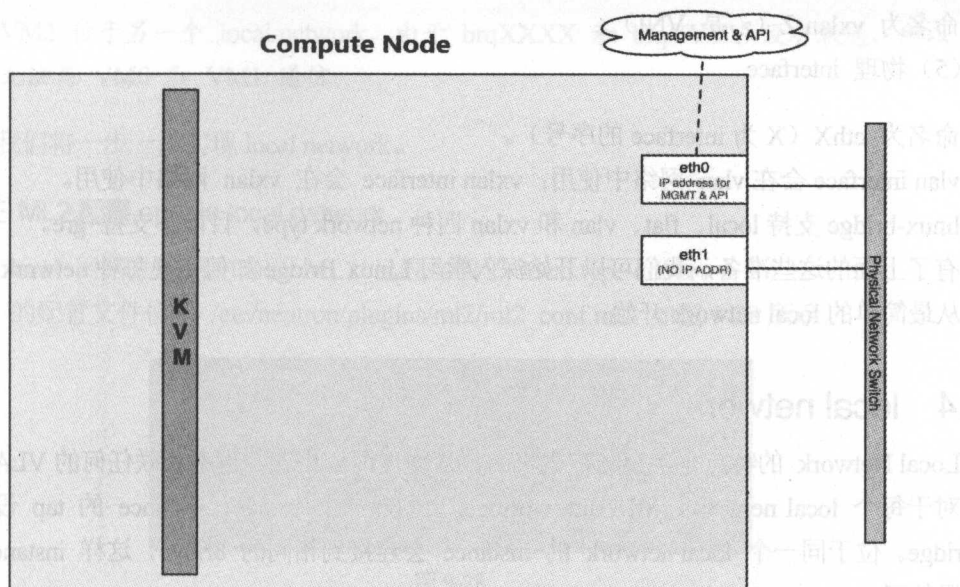


图 9-25

```

root@devstack-compute1:~# ifconfig
eth0      Link encap:Ethernet  Hwaddr 00:50:56:b0:a2:ba
          inet addr:192.168.104.11  Bcast:192.168.104.255  Mask:255.255.255.0
          inet6 addr: fe80::250:56ff:feb0:a2ba/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:117 errors:0 dropped:7 overruns:0 frame:0
          TX packets:120 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:14807 (14.8 KB)  TX bytes:17250 (17.2 KB)

eth1      Link encap:Ethernet  Hwaddr 00:50:56:b0:b5:40
          inet6 addr: fe80::250:56ff:feb0:b540/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:10 errors:0 dropped:7 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:600 (600.0 B)  TX bytes:798 (798.0 B)
  
```

图 9-26

### 9.4.3 了解 Linux Bridge 环境中的各种网络设备

在配置 linux bridge driver 之前先了解几种网络设备，后面会经常用到。

在 Linux Bridge 环境中，一个数据包从 instance 发送到物理网卡会经过下面几个类型的设备：

#### (1) tap interface

命名为 tapN (N 为 0、1、2、3.....)。

#### (2) Linux Bridge

命名为 brqXXXX。

#### (3) vlan interface

命名为 ethX.Y (X 为 interface 的序号，Y 为 vlan id)。

#### (4) vxlan interface

命名为 vxlan-Z (z 是 VNI)。

#### (5) 物理 interface

命名为 ethX (X 为 interface 的序号)。

vlan interface 会在 vlan 网络中使用; vxlan interface 会在 vxlan 网络中使用。

linux-bridge 支持 local、flat、vlan 和 vxlan 四种 network type, 目前不支持 gre。

有了上面的这些准备, 我们可以开始深入学习 Linux Bridge 如何实现每种 network type 了。首先从最简单的 local network 开始。

### 9.4.4 local network

Local Network 的特点是不会与宿主机的任何物理网卡相连, 也不关联任何的 VLAN ID。

对于每个 local network, ML2 linux-bridge 会创建一个 bridge, instance 的 tap 设备会连接到 bridge。位于同一个 local network 的 instance 会连接到相同的 bridge, 这样 instance 之间就可以通信了。

因为 bridge 没有与物理网卡连接, 所以 instance 无法与宿主机之外的网络通信。

同时因为每个 local network 有自己的 bridge, bridge 之间是没有连通的, 所以两个 local network 之间也不能通信, 即使它们位于同一宿主机上。

图 9-27 是 local network 的示例:

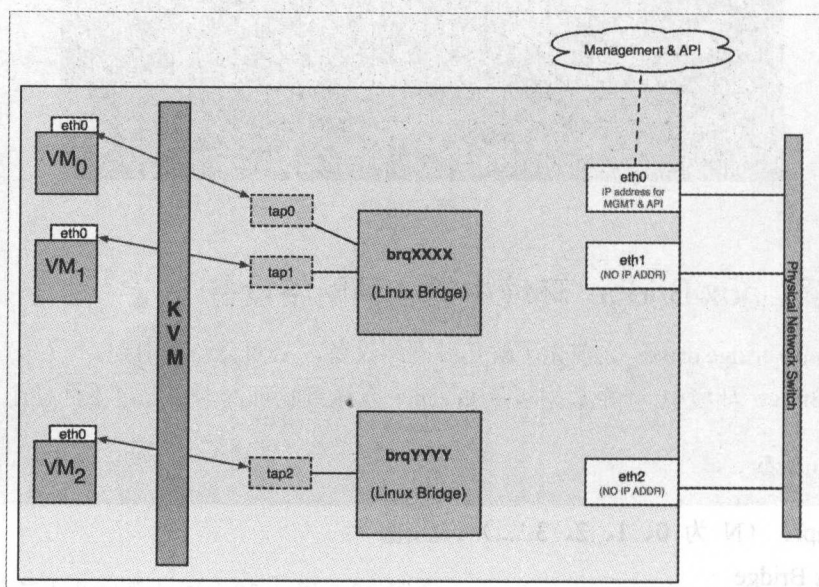


图 9-27

- 创建了两个 local network, 分别对应两个网桥 brqXXXX 和 brqYYYY。
- VM0 和 VM1 通过 tap0 和 tap1 连接到 brqXXXX。
- VM2 通过 tap0 和 tap2 连接到 brqYYYY。
- VM0 与 VM1 在同一个 local network 中, 它们之间可以通信。

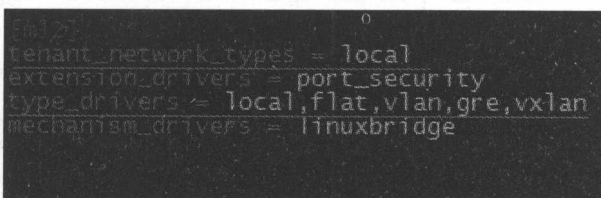
- VM2 位于另一个 local network, 由于 brqXXXX 和 brqYYYY 没有联通, 所以 VM2 无法与 VM0 和 VM1 通信。

下面我们将一步一步实现 local network。

### 1. 在 ML2 配置 enable local network

创建 local 网络之前请先确保 ML2 已经加载了 local type driver。

ML2 的配置文件位于 /etc/neutron/plugins/ml2/ml2\_conf.ini。如图 9-28 所示。



```
tenant_network_types = local
extension_drivers = port_security
type_drivers = local,flat,vlan,gre,vxlan
mechanism_drivers = linuxbridge
```

图 9-28

type\_drivers 告诉 ML2 加载所有 5 种网络的 type driver。

```
type_drivers = local,flat,vlan,gre,vxlan
```

这样所有类型的网络我们都可以创建（虽然在本节只创建 local 网络）。

普通用户和 admin 都可以通过 CLI 或者 Web GUI 创建网络, 但只有 admin 才能指定网络的 type, 所以需要用到 tenant\_network\_types 告诉 ML2 当普通用户在自己的 Tenant (Project) 中创建网络时, 默认创建哪种 type 的 7F51 网, 这里 type 是 local。

```
tenant_network_types = local
```

tenant\_network\_types 可以指定多种 type, 比如:

```
tenant_network_types = vlan, local
```

其作用是先创建 vlan 网络, 当没有 vlan 可创建时 (比如 vlan id 用完), 便创建 local 网络。当配置文件发生了变化, 需要重启 Neutron 相关服务使之生效。

### 2. 创建第一个 local network

下面我们通过 Web GUI 创建第一个 local network。

首先确保各个节点上的 neutron agent 状态正常。GUI 中执行菜单命令 Admin → System → System Information → Neutron Agents, 如图 9-29 所示。

admin

System Information

Services   Compute Services   Block Storage Services   Network Agents

Type	Name	Host	Status	State
DHCP agent	neutron-dhcp-agent	devstack-controller	Enabled	Up
Linux bridge agent	neutron-linuxbridge-agent	devstack-controller	Enabled	Up
Loadbalancer agent	neutron-lbaas-agent	devstack-controller	Enabled	Up
L3 agent	neutron-vpn-agent	devstack-controller	Enabled	Up
Linux bridge agent	neutron-linuxbridge-agent	devstack-compute1	Enabled	Up
Metadata agent	neutron-metadata-agent	devstack-controller	Enabled	Up

Displaying 6 items

图 9-29

GUI 中有两个地方可以创建 network:

(1) Project → Network → Networks

这是普通用户在自己的 tenant 中创建 network 的地方。

(2) Admin → Networks

这是 admin 创建 network 的地方。

我们先用第一种方式创建，单击“Create Network”按钮，如图 9-30 所示。

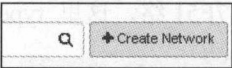


图 9-30

在创建网络的向导页面，把 network 命名为“first\_local\_net”，如图 9-31 所示。

Create Network

Network   Subnet   Subnet Details

Network Name

first\_local\_net

Create a new network. In addition, a subnet associated with the network can be created in the next panel.

Admin State

UP

☒ Create Subnet

Cancel   < Back   Next >

图 9-31

单击“Next”，创建 subnet，命名为“subnet\_172\_16\_1\_0”，地址为“172.16.1.0/24”。



如果 Gateway IP 不设置，默认为 subnet 的第一个 IP，即 172.16.1.1，如图 9-32 所示。

**Create Network**

Subnet Subnet Details

Subnet Name  
subnet\_172\_16\_1\_0

Create a subnet associated with the network.  
Advanced configuration is available by clicking on the "Subnet Details" tab.

Network Address  
172.16.1.0/24

IP Version  
IPv4

Gateway IP

☐ Disable Gateway

Cancel < Back Next >

图 9-32

单击“Next”，设置 subnet 的 IP 地址范围为 172.16.1.2-172.16.1.100，instance 的 IP 会从这里分配。

默认会选中“Enable DHCP”，同时还可以设置 subnet 的 DNS 和添加静态路由条目，如图 9-33 所示。

**Create Network**

Subnet Subnet Details

☒ Enable DHCP

Specify additional attributes for the subnet

Allocation Pools  
172.16.1.2, 172.16.1.100

DNS Name Servers

Host Routes

< Back Create

图 9-33

单击“Create”，network 创建成功。如图 9-34 所示。

Networks				
<div>Create Network</div>				<div>Filter</div>
<input type="checkbox"/>	Name	Subnets Associated	Shared	Status
<input type="checkbox"/>	first_local_net	subnet_172_16_1_0 172.16.1.0/24	No	Active
Displaying 1 item				

图 9-34

底层网络发生了什么变化

Network“first\_local\_net”已经创建成功了，下面我们需要搞清楚底层网络结构有了哪些变化？单击“first\_local\_net”链接，显示 network 的 subnet 和 port 信息，如图 9-35 所示。

Subnets

<input type="checkbox"/>	Name	Network Address	IP Version	Gateway IP
<input type="checkbox"/>	subnet_172_16_1_0	172.16.1.0/24	IPv4	172.16.1.1

Displaying 1 item

Ports

Name	Fixed IPs	Attached Device	Status	Admin State
(a5bd3746-3f89)	172.16.1.2	networkdhcp	Active	UP

Displaying 1 item

图 9-35

在 Ports 列表中已经创建了一个 port，名称为“(a5bd3746-3f89)”，IP 为 172.16.1.2，Attached Device 是 network:dhcp。

这里我们只需要知道该 port 对应的是 dhcp 的 interface，至于 dhcp 如何工作后面有专门的章节详细讨论。

打开控制节点的 shell 终端，用 brctl show 查看当前 linux bridge 的状态，如图 9-36 所示。

```
root@devstack-controller:~# brctl show
bridge name      bridge id        STP enabled      interfaces
brqbb9b6d21-c6   8000:622719b08e6d no                tapa5bd3746-3f
virbr0           8000:000000000000 yes
```

图 9-36

可以看到 Neutron 自动创建了如下两个设备：

- bridge 设备 brqbb9b6d21-c6

brqbb9b6d21-c6 对应 local network“first\_local\_net”，命名规则为 brqXXX，XXX 为 network ID 的前 11 个字符。

- tap 设备 tapa5bd3746-3f

tapa5bd3746-3f 对应 port (a5bd3746-3f89)，命名规则为 tapYYY，YYY 为 port ID 的前 11 个字符。

该 tap 设备已经连接到 bridge，即连接到该 flat 网络。

### 3. 将 instance 连接到 first\_local\_net

first\_local\_net 已经就绪，下面创建 instance 并将其连接到该网络。

launch 一个 instance，在“Networking”标签页面选择 first\_local\_net 网络，如图 9-37 所示。

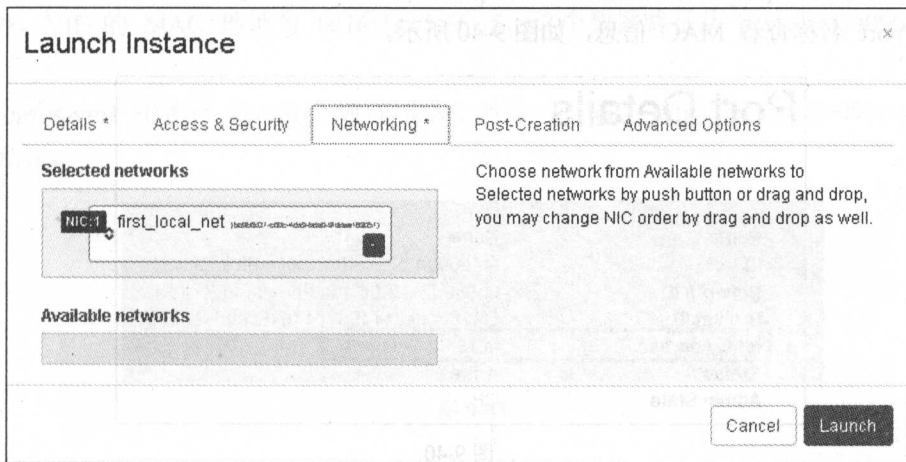


图 9-37

instance 部署成功，分配的 IP 地址为 172.16.1.3，如图 9-38 所示。

<input type="checkbox"/>	Instance Name	Image Name	IP Address
<input type="checkbox"/>	cirros-vm1	cirros	172.16.1.3
Displaying 1 item			

图 9-38

底层网络发生了什么变化

对于 instance “cirros-vm1”，Neutron 会在 subnet 中创建一个 port，分配 IP 和 MAC 地址，并将 port 分配给 cirros-vm1，如图 9-39 所示。

Ports		
Name	Fixed IPs	Attached Device
(a5bd3746-3f89)	172.16.1.2	network:dhcp
(fa7e090e-a29c)	172.16.1.3	compute:nova
Displaying 2 items		

图 9-39

如图 9-39 所示，port 列表中增加了一个 port “(fa7e090e-a29c)”，IP 为 172.16.1.3。单击 port 名称查看 MAC 信息，如图 9-40 所示。

Port Details	
Name	None
ID	fa7e090e-a29c-40c7-a5ce-9fca88f0c314
Network ID	bb9b6d21-c60c-4da0-bda0-3fddee169051
Project ID	5f476b88c2414932b17648cebd45a97c
MAC Address	fa:16:3e:c1:86:a5
Status	Active
Admin State	UP

图 9-40

当 cirros-vm1 启动时：

- (1) 宿主机上的 neutron-linuxbridge-agent 会根据 port 信息创建 tap 设备，并连接到 local 网络所在的 bridge。
- (2) 同时该 tap 会映射成 cirros-vm1 的虚拟网卡，即 virtual interface（VIF）。

下面我们验证一下以上信息：

cirros-vm1 部署到了控制节点，通过 brctl show 查看 bridge 的配置，如图 9-41 所示。

```
root@devstack-controller:~#  
root@devstack-controller:~# brctl show  
bridge name    bridge id        STP enabled    interfaces  
brqbb9b6d21-c6  8000.622719b08e6d  no             tapfa5bd3746-3f  
virbr0         8000.000000000000  yes            tapfa7e090e-a2  
root@devstack-controller:~#  
root@devstack-controller:~#  
root@devstack-controller:~# virsh list  
Id      Name              State  
-----  
2       instance-00000001  running  
root@devstack-controller:~#
```

图 9-41

可以看到 bridge brqbb9b6d21-c6 上连接了一个新的 tap 设备 tapfa7e090e-a2。从命名上可知



tapfa7e090e-a2 对应着 port “(fa7e090e-a29c)”。

virsh list 中显示的虚拟机 instance-00000001 即为“cirros-vm1”，命名方式有所不同，需注意。通过 virsh edit 命令查看 cirros-vm1 的配置，确认 VIF 就是 tapfa7e090e-a2，如图 9-42 所示。

```
<interface type=
  <mac address=
  <source bridge=
  <target dev=
  <model type=
  <driver name=
  <address type= domain= bus=
  </interface>
```

图 9-42

另外，VIF 的 MAC 地址为 fa:16:3e:c1:66:a5，这个数据就是从 port “(fa7e090e-a29c)” 取过来的。

在 cirros-vm1 中执行 ifconfig，通过 MAC 地址可以确认 eth0 与 tapfa7e090e-a2 对应，如图 9-43 所示。

```
ifconfig
eth0: Link encap:Ethernet HWaddr fa:16:3e:c1:66:a5
      inet addr:172.16.1.3 Bcast:172.16.1.255 Mask:255.255.255
      inet6 addr: fe80::f816:3eff:fecl:66a5/64 Scope:Link
      UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
      RX packets:26 errors:0 dropped:0 overruns:0 frame:0
      TX packets:32 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:1000
      RX bytes:2687 (2.6 KiB) TX bytes:2810 (2.7 KiB)
```

图 9-43

图 9-44 展示了创建 cirros-vm1 后宿主机当前的网络结构。

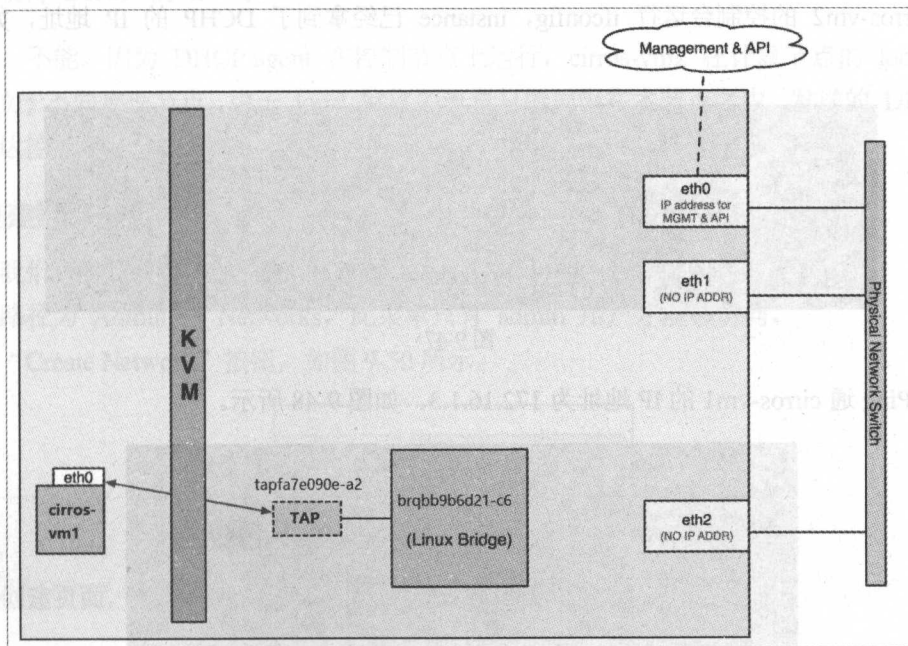


图 9-44

4. 连接第二个 instance 到 first\_local\_net

以同样的方式 launch instance “cirros-vm2”。

分配的 IP 为 172.16.1.4，如图 9-45 所示。

<input type="checkbox"/>	Instance Name	Image Name	IP Address
<input type="checkbox"/>	cirros-vm2	cirros	172.16.1.4
<input type="checkbox"/>	cirros-vm1	cirros	172.16.1.3
Displaying 2 items			

图 9-45

cirros-vm2 也被 schedule 到控制节点，virsh list 和 brctl show 输出如图 9-46 所示。  
cirros-vm2 对于 tap 的设备为 tapa5bd3746-3f。

```
root@devstack-controller:~# brctl show
bridge name      bridge id        STP enabled      interfaces
brqbb9b6d21-c6   8000.622719b08e6d  no               tap2fd559e1-4d
                                                           tapa5bd3746-3f
                                                           tapfa7e090e-a2

virbr0            8000.000000000000  yes
root@devstack-controller:~# virsh list
 Id    Name                                     State
-----
 2     instance-00000001                       running
 4     instance-00000002                       running
```

图 9-46

在 cirros-vm2 的控制台运行 ifconfig，instance 已经拿到了 DHCP 的 IP 地址，如图 9-47 所示。

```
$ ifconfig
eth0      Link encap:Ethernet  HWaddr FA:16:3E:DA:F7:E1
          inet addr:172.16.1.4  Bcast:172.16.1.255  Mask:255.255.255.0
          inet6 addr: fe80::f816:3eff:fedd:f7e1/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:23 errors:0 dropped:0 overruns:0 frame:0
          TX packets:92 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2359 (2.3 KiB)  TX bytes:5128 (5.0 KiB)
```

图 9-47

能够 Ping 通 cirros-vm1 的 IP 地址为 172.16.1.3，如图 9-48 所示。

```
$ ping 172.16.1.3
PING 172.16.1.3 (172.16.1.3): 56 data bytes
64 bytes from 172.16.1.3: seq=0 ttl=64 time=1.699 ms
64 bytes from 172.16.1.3: seq=1 ttl=64 time=1.190 ms
64 bytes from 172.16.1.3: seq=2 ttl=64 time=1.357 ms
64 bytes from 172.16.1.3: seq=3 ttl=64 time=1.286 ms
64 bytes from 172.16.1.3: seq=4 ttl=64 time=1.796 ms
64 bytes from 172.16.1.3: seq=5 ttl=64 time=1.426 ms
64 bytes from 172.16.1.3: seq=6 ttl=64 time=1.581 ms
64 bytes from 172.16.1.3: seq=7 ttl=64 time=1.056 ms
```

图 9-48

当前宿主机的网络结构如图 9-49 所示。

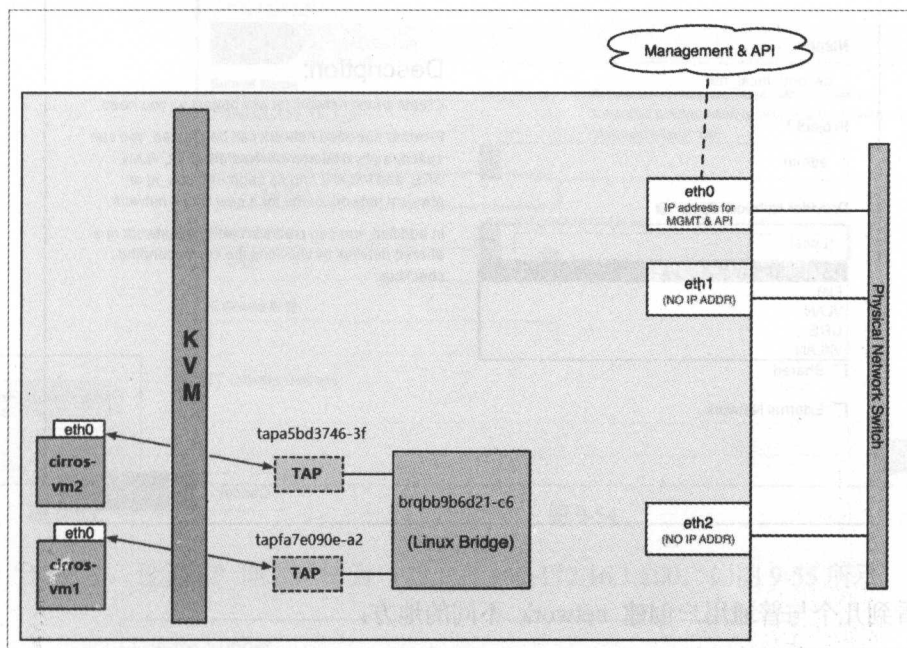


图 9-49

两个 instance 的 VIF 挂在同一个 Linux Bridge 上，可以相互通信。

这里请大家思考一个问题：如果 cirros-vm2 launch 时被 schedule 到计算节点而非控制节点，它能获得 DHCP 的 IP 吗？

答案：不能。因为 DHCP agent 在控制节点上运行，cirros-vm2 在计算节点的 local 网络上，两者位于不同物理节点。由于 local 网络的流量只能局限在本节点之内，发送的 DHCP 请求无法到达控制节点。

## 5. 创建第二个 local network

本节我们以第二种方式创建 local network “second\_local\_net”。

菜单路径为 Admin → Networks，此菜单只有 admin 用户才能够访问。

单击“Create Network”按钮，如图 9-50 所示。

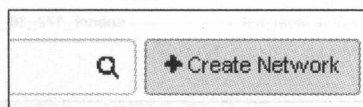


图 9-50

显示创建页面，如图 9-51 所示。

Create Network

Name

second\_local\_net

Project \*

admin

Provider Network Type \* ?

Local

Local

Flat

VLAN

GRE

VXLAN

☐ Shared

☐ External Network

Description:

Create a new network for any project as you need.  
Provider specified network can be created. You can specify a physical network type (like Flat, VLAN, GRE, and VXLAN) and its segmentation\_id or physical network name for a new virtual network.  
In addition, you can create an external network or a shared network by checking the corresponding checkbox.

Cancel

Create Network

图 9-51

可以看到几个与普通用户创建 network 不同的地方：

- 可以选择该 network 属于哪个 Project（租户）。
- 可以选择 network type。
- 可以指定 network 是否与其他 Project 共享。
- 可以指定是否为 external network。

可见，这种方式赋予 admin 用户创建 network 更大的灵活性，后面我们都将采用这种方式创建 network。

单击“Create Network”，second\_local\_net 创建成功，如图 9-52 所示。

Networks

<input type="checkbox"/>	Project	Network Name	Subnets Associated
<input type="checkbox"/>	admin	second_local_net	
<input type="checkbox"/>	admin	first_local_net	subnet_172.16.1.0/24 172.16.1.0/24

Displaying 2 items

图 9-52

单击“second\_local\_net”链接，进入 network 配置页面，目前还没有 subnet，单击“Create Subnet”按钮，如图 9-53 所示。

设置 IP 地址为“172.16.1.0/24”，如图 9-54 所示。



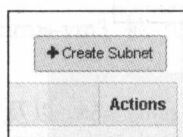


图 9-53

**Create Subnet**

Subnet Subnet Details

**Subnet Name**  
subnet\_172.16.1\_0

Create a subnet associated with the network.  
Advanced configuration is available by clicking on the "Subnet Details" tab.

**Network Address**  
172.16.1.0/24

**IP Version**  
IPv4

**Gateway IP**

☐ Disable Gateway

« Back Next »

图 9-54

单击“Next”，设置 IP 地址范围为 172.16.1.101-172.16.1.200，如图 9-55 所示。

**Create Subnet**

Subnet Subnet Details

☒ Enable DHCP

Specify additional attributes for the subnet.

**Allocation Pools**  
172.16.1.101, 172.16.1.200

**DNS Name Servers**

**Host Routes**

« Back Create

图 9-55

单击“Create”，subnet 创建成功，如图 9-56 所示。

### Subnets

<input type="checkbox"/>	Name	CIDR
<input type="checkbox"/>	subnet_172_16_1_0	172.16.1.0/24

Displaying 1 item

### Ports

<input type="checkbox"/>	Name	Fixed IPs	Attached Device
<input type="checkbox"/>	(ae547b6b-2aab)	172.16.1.101	network:dhcp

Displaying 1 item

图 9-56

查看控制节点的网络结构，增加了 second\_local\_net 对应的网桥 brq161e0b25-58，以及 dhcp 的 tap 设备 tapae547b6b-2aa，如图 9-57 所示。

```
root@devstack-controller:~# brctl show
bridge name      bridge id        STP enabled     interfaces
brq161e0b25-58    8000.568da583b1af  no              tapae547b6b-2a
brq6b9b6d21-c6    8000.622719b08e6d  no              tapa5bd3746-3f
virbr0            8000.000000000000  yes             tapfa7e090e-a2
root@devstack-controller:~#
```

图 9-57

6. 将 instance 连接到 second\_local\_net

launch 新的 instance “cirros-vm3”，网络选择 second\_local\_net，如图 9-58 所示。

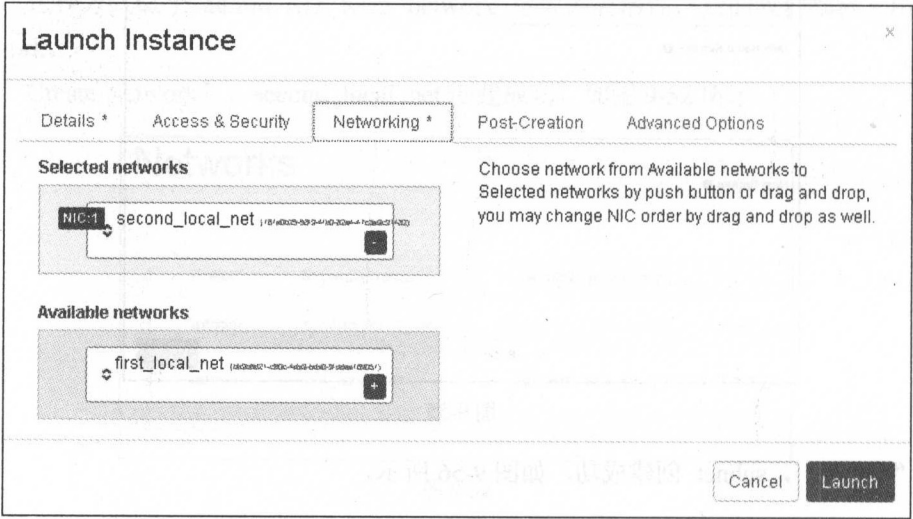


图 9-58

cirros-vm3 分配到的 IP 为 172.16.1.102，如图 9-59 所示。

<input type="checkbox"/>	Instance Name	Image Name	IP Address
<input type="checkbox"/>	cirros-vm3	cirros	172.16.1.102
<input type="checkbox"/>	cirros-vm2	cirros	172.16.1.4
<input type="checkbox"/>	cirros-vm1	cirros	172.16.1.3
Displaying 3 items			

图 9-59

cirros-vm3 被 schedule 到控制节点, 对应的 tap 设备为 tap5395d19b-ed, 如图 9-60 所示。

```

root@devstack-controller:~# virsh list
-----
 Id       Name                                     State
-----
 2        instance-00000001                      running
 4        instance-00000002                      running
 5        instance-00000004                      running

root@devstack-controller:~# brctl show
bridge name      bridge id        STP enabled    interfaces
brq161e0b25-58   8000.568da583b1af no              tap5395d19b-ed
brqbb9b6d21-c6   8000.622719b08e6d no              tapae547b6b-2a
virbr0           8000.000000000000 yes             tap2fd559e1-4d
                                                         tapa5bd3746-3f
                                                         tapfa7e090e-a2
root@devstack-controller:~#

```

图 9-60

控制台显示 cirros-vm3 已经成功地从 DHCP 拿到 IP 地址 172.16.1.102, 如图 9-61 所示。

```

root@devstack-controller:~# ifconfig eth0
eth0: Link encap:Ethernet HWaddr FA:16:3E:5B:BF:00
       inet addr:172.16.1.102 Bcast:172.16.1.255 Mask:255.255.255.0
       inet6 addr: fe80::f816:3eff:fe5b:bf00/64 Scope:Link
       UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
       RX packets:34 errors:0 dropped:0 overruns:0 frame:0
       TX packets:49 errors:0 dropped:0 overruns:0 carrier:0
       collisions:0 txqueuelen:1000
       RX bytes:3325 (3.2 KiB) TX bytes:4112 (4.0 KiB)

```

图 9-61

但是 cirros-vm3 无法 Ping 到 cirros-vm1, 如图 9-62 所示。

```

root@devstack-controller:~# ping 172.16.1.3
PING 172.16.1.3 (172.16.1.3): 56 data bytes

```

图 9-62

这是在预料之中的, 因为 cirros-vm3 和 cirros-vm1 位于不同的 local network, 之间没有连通, 即使都位于同一个宿主机也不能通信。

网络结构如图 9-63 所示。

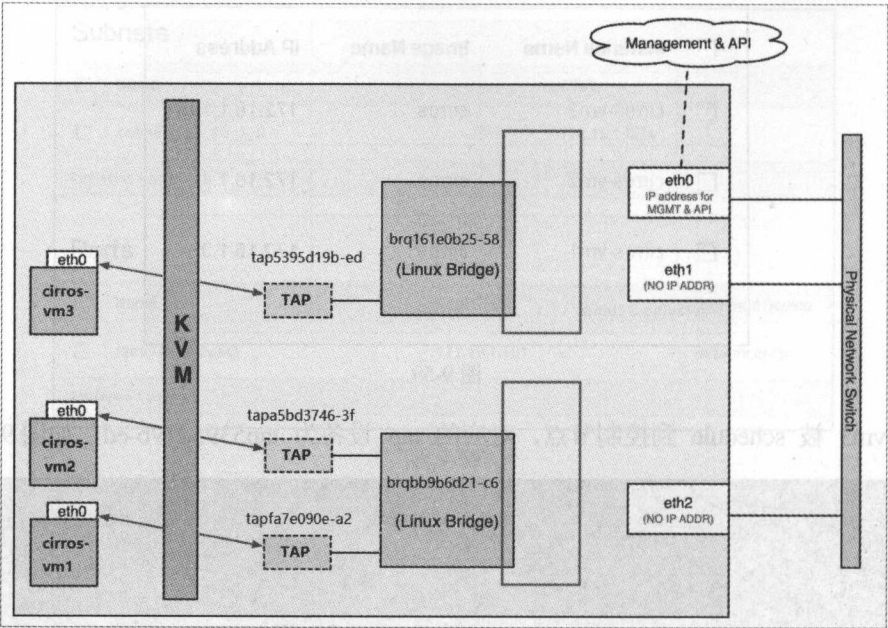


图 9-63

### 9.4.5 flat network

flat network 是不带 tag 的网络，要求宿主机的物理网卡直接与 Linux Bridge 连接，这意味着每个 flat network 都会独占一个物理网卡，如图 9-64 所示。

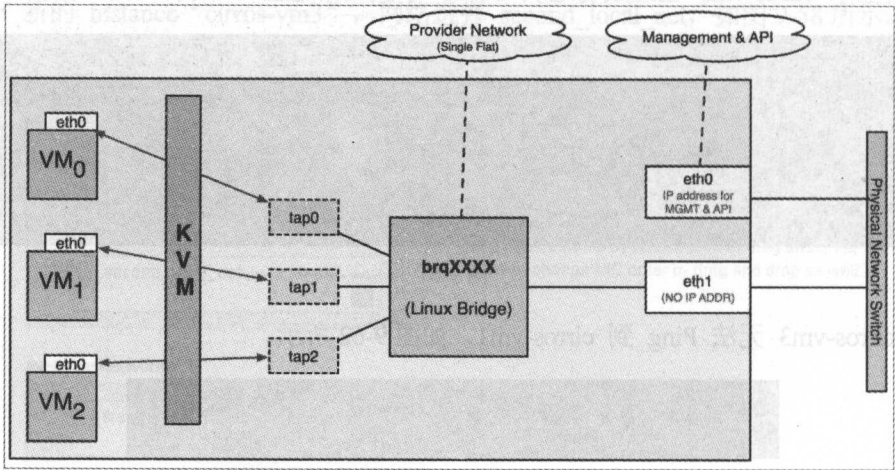


图 9-64

图 9-64 中 eth1 桥接到 brqXXXX，为 instance 提供 flat 网络。如果需要创建多个 flat network，就得准备多个物理网卡，如图 9-65 所示。



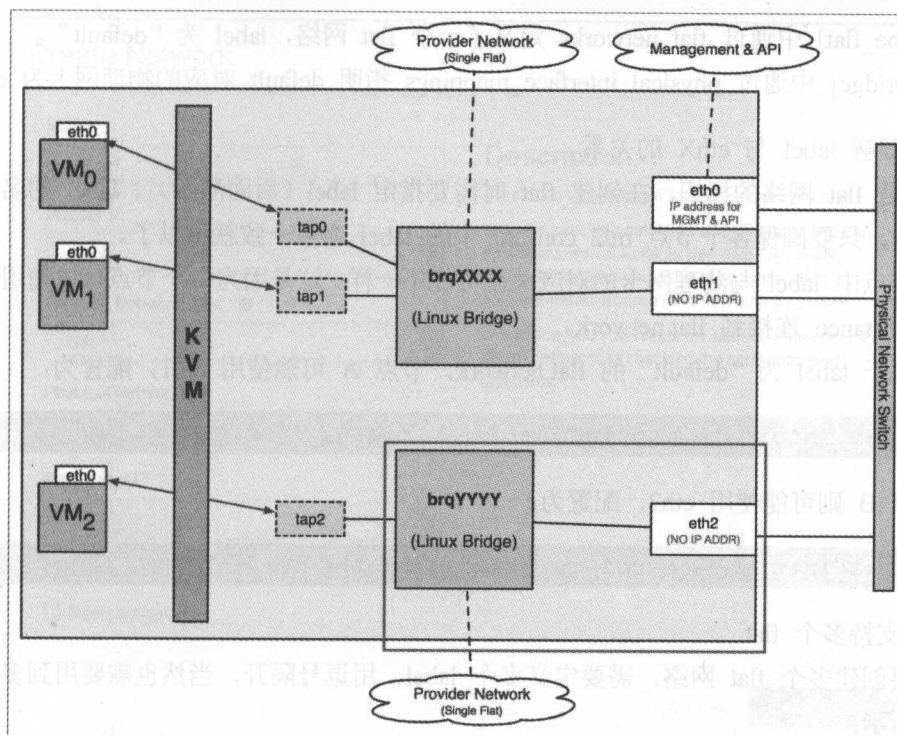


图 9-65

接下来配置 flat 网络。

### 1. 在 ML2 中配置 enable flat network

在 `/etc/neutron/plugins/ml2/ml2_conf.ini` 设置 flat network 相关参数，如图 9-66 所示。

```
tenant_network_types = flat
extension_drivers = port_security
type_drivers = local,flat,vlan,gre,vxlan
mechanism_drivers = linuxbridge
```

图 9-66

```
tenant_network_types = flat
```

指定普通用户创建的网络类型为 flat。需要注意的是：因为 flat 网络与物理网卡一一对应，一般情况下租户网络不会采用 flat，这里只是示例。

接着需要指明 flat 网络与物理网卡的对应关系，如图 9-67 所示。

```
default_bridge = default
```

```
physical_interface_mappings = default:eth1
```

图 9-67

如上所示：

在 [ml2\_type\_flat] 中通过 flat\_networks 定义了一个 flat 网络, label 为 “default”。

在 [linux\_bridge] 中通过 physical\_interface\_mappings 指明 default 对应的物理网卡为 eth1。

#### (1) 理解 label 与 ethX 的关系

label 是 flat 网络的标识, 在创建 flat 时需要指定 label (后面演示)。label 的名字可以是任意字符串, 只要确保各个节点 ml2\_conf.ini 中的 label 命名一致就可以了。

各个节点中 label 与物理网卡的对应关系可能不一样。这是因为每个节点可以使用不同的物理网卡将 instance 连接到 flat network。

例如对于 label 为 “default” 的 flat network, 节点 A 可能使用 eth1, 配置为:

```
physical_interface_mappings = default:eth1
```

而节点 B 则可能使用 eth2, 配置为:

```
physical_interface_mappings = default:eth2
```

#### (2) 支持多个 flat

如果要创建多个 flat 网络, 需要定义多个 label, 用逗号隔开, 当然也需要用到多个物理网卡, 如下所示:

```
[ml2_type_flat]
flat_networks = flat1,flat2
[linux_bridge]
physical_interface_mappings = flat1:eth1,flat2:eth2
```

准备就绪, 可以创建 flat 网络了。

## 2. 创建 flat network “flat\_net”

打开菜单 Admin → Networks, 单击 “Create Network” 按钮, 如图 9-68 所示。

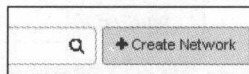


图 9-68

显示创建页面, 如图 9-69 所示。

### Create Network

**Name**

**Project \***

**Provider Network Type \* ?**

**Physical Network \* ?**

**Admin State \***

☐ Shared

☐ External Network

Cancel Create Network

**Description:**

Create a new network for any project as you need.

Provider specified network can be created. You can specify a physical network type (like Flat, VLAN, GRE, and VXLAN) and its segmentation\_id or physical network name for a new virtual network.

In addition, you can create an external network or a shared network by checking the corresponding checkbox.

图 9-69

Provider Network Type 中选择“Flat”。

Physical Network 中填写“default”，与 ml2\_conf.ini 中 flat\_networks 参数保持一致。

单击“Create Network”，flat\_net 创建成功，如图 9-70 所示。

<input type="checkbox"/>	Project	Network Name	Subnets Associated
<input type="checkbox"/>	admin	flat_net	
Displaying 1 item			

图 9-70

单击 flat\_net 链接，进入 network 配置页面，目前还没有 subnet，单击“Create Subnet”按钮，如图 9-71 所示。

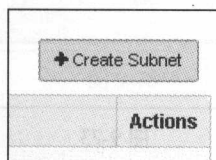


图 9-71

设置 IP 地址为“172.16.1.0/24”，如图 9-72 所示。

**Create Subnet**

Subnet Subnet Details

Subnet Name  
subnet\_172\_16\_1\_0

Network Address ?  
172.16.1.0/24

IP Version  
IPv4

Gateway IP ?

☐ Disable Gateway

« Back Next »

Create a subnet associated with the network.  
Advanced configuration is available by clicking on the "Subnet Details" tab.

图 9-72

单击“Next”，设置 IP 地址范围为 172.16.1.101-172.16.1.200，如图 9-73 所示。

**Create Subnet**

Subnet Subnet Details

☒ Enable DHCP

Specify additional attributes for the subnet.

Allocation Pools ?  
172.16.1.101,172.16.1.200

DNS Name Servers ?

Host Routes ?

« Back Create

图 9-73

单击“Create”，subnet 创建成功，如图 9-74 所示。



### Subnets

<input type="checkbox"/>	Name	CIDR
<input type="checkbox"/>	subnet_172_16_1_0	172.16.1.0/24

Displaying 1 item

### Ports

<input type="checkbox"/>	Name	Fixed IPs	Attached Device
<input type="checkbox"/>	(19a0ed3d-fe20)	172.16.1.102	network:dhcp

Displaying 1 item

图 9-74

(1) 底层网络发生了什么变化

执行 `brctl show`，查看控制节点当前的网络结构，如图 9-75 所示。

```

root@devstack-controller:~# brctl show
bridge name      bridge id        STP enabled     interfaces
brqf153b42f-c3   8000.005056b0b5ff no               eth1
vibrbr0          8000.000000000000 yes              tap19a0ed3d-fe
root@devstack-controller:~#

```

图 9-75

Neutron 自动新建了 `flat_net` 对应的网桥 `brqf153b42f-c3`，以及 `dhcp` 的 `tap` 设备 `tap19a0ed3d-fe`。

另外，`tap19a0ed3d-fe` 和物理网卡 `eth1` 都已经连接到 `bridge`。

此时 `flat_net` 结构如图 9-76 所示。

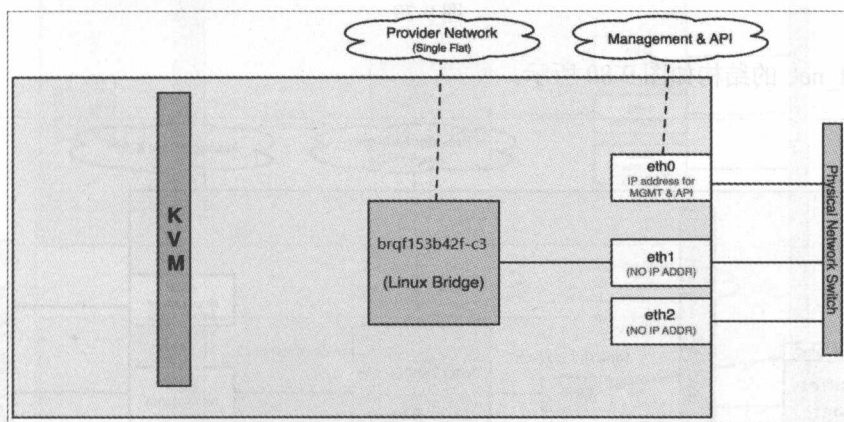


图 9-76

(2) 将 `instance` 连接到 `flat_net`

launch 新的 `instance` “`cirros-vm1`”，选择网络 `flat_net`，如图 9-77 所示。

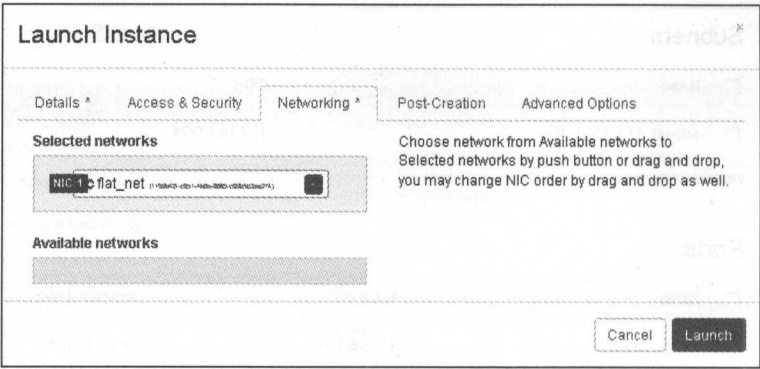


图 9-77

cirros-vm1 分配到的 IP 为 172.16.1.103，如图 9-78 所示。

<input type="checkbox"/>	Instance Name	Image Name	IP Address
<input type="checkbox"/>	cirros-vm1	cirros	172.16.1.103
Displaying 1 item.			

图 9-78

cirros-vm1 被 schedule 到控制节点，对应的 tap 设备为 tapc1875c7f-cb，并且已经连接到 bridge，如图 9-79 所示。

```
root@devstack-controller:~# brctl show
bridge name      bridge id        STP enabled  interfaces
brqf153b42f-c3   8000.005056b0b5ff  no           eth1
                                                         tap19a0ad1d-fe
                                                         tapc1875c7f-cb
virbr0           8000.000000000000  yes
```

图 9-79

当前 flat\_net 的结构如图 9-80 所示。

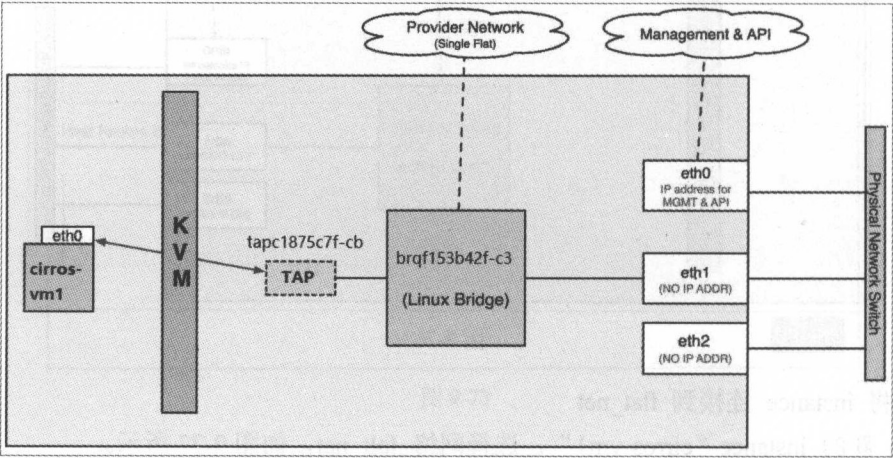


图 9-80

继续用同样的方式 launch instance cirros-vm2, 分配到的 IP 为 172.16.1.104, 如图 9-81 所示。

<input type="checkbox"/>	Instance Name	Image Name	IP Address
<input type="checkbox"/>	cirros-vm2	cirros	172.16.1.104
<input type="checkbox"/>	cirros-vm1	cirros	172.16.1.103

Displaying 2 Items

图 9-81

cirros-vm2 被 schedule 到计算节点, 对应的 tap 设备为 tapfb3fb197-24, 并且连接到 bridge, 如图 9-82 所示。

```

root@devstack-compute1:~# brctl show
bridge name      bridge id        STP enabled      interfaces
brqf153b42f-c3   8000.005056b0b540  no               eth1
                                                           tapfb3fb197-24
virbr0           8000.000000000000  yes
root@devstack-compute1:~#

```

图 9-82

这里有两点需要提醒:

- 因为计算节点上没有 HDPCP 服务, 所以 brctl show 中没有 DHCP 对应的 tap 设备。
- 计算节点上 bridge 的名称与控制节点上一致, 都是 brqf153b42f-c3, 表明是同一个 network。

当前 flat\_net 的结构如图 9-83 所示。

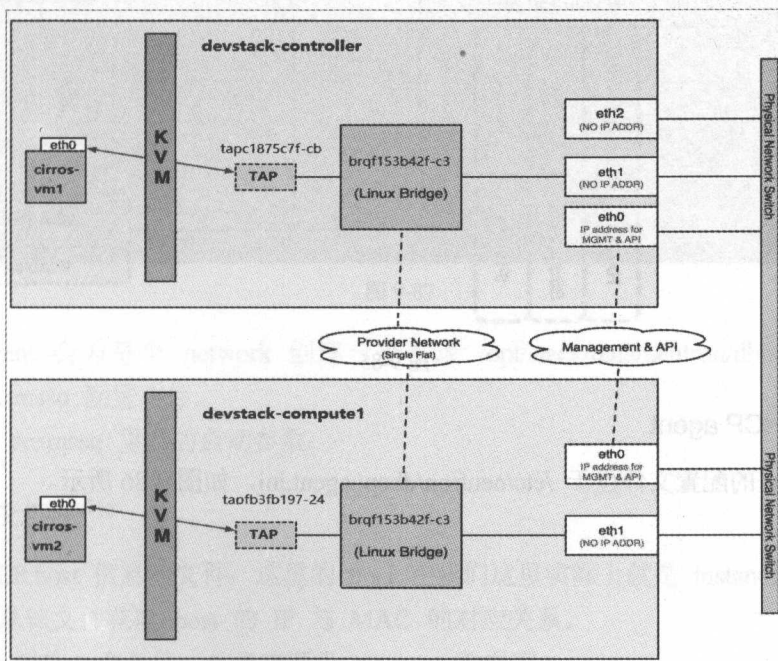


图 9-83

cirros-vm1 (172.16.1.103) 与 cirros-vm2 (172.16.1.104) 位于不同节点, 通过 flat\_net 相连, 下面执行 PING 验证连通性。

在 cirros-vm1 控制台中执行 ping 172.16.1.104, 如图 9-84 所示。

```
$ ping 172.16.1.104
PING 172.16.1.104 (172.16.1.104): 56 data bytes
64 bytes from 172.16.1.104: seq=0 ttl=64 time=2.078 ms
64 bytes from 172.16.1.104: seq=1 ttl=64 time=2.120 ms
64 bytes from 172.16.1.104: seq=2 ttl=64 time=3.503 ms
64 bytes from 172.16.1.104: seq=3 ttl=64 time=1.474 ms
64 bytes from 172.16.1.104: seq=4 ttl=64 time=1.949 ms
64 bytes from 172.16.1.104: seq=5 ttl=64 time=1.439 ms
```

图 9-84

如我们预料, ping 成功。

9.4.6 DHCP 服务

前面章节我们看到 instance 在启动过程中能够从 Neutron 的 DHCP 服务获得 IP, 本节将详细讨论其内部实现机制。

Neutron 提供 DHCP 服务的组件是 DHCP agent。

DHCP agent 在网络节点运行上, 默认通过 dnsmasq 实现 DHCP 功能, 如图 9-85 所示。

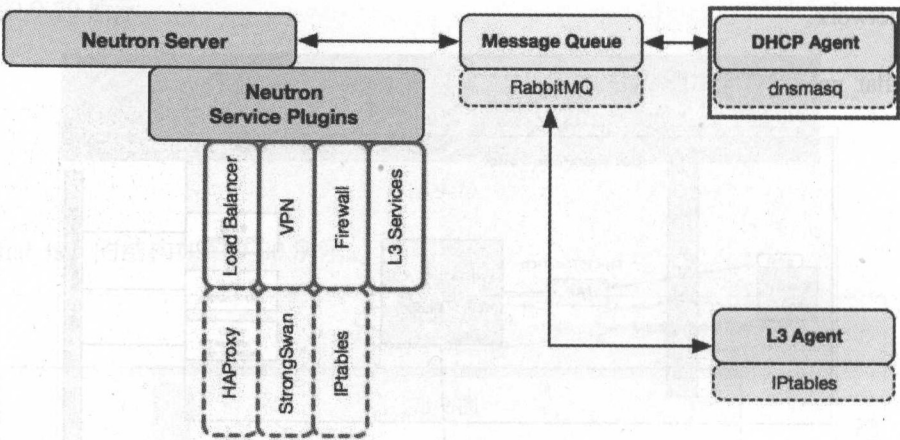


图 9-85

1. 配置 DHCP agent

DHCP agent 的配置文件位于 /etc/neutron/dhcp\_agent.ini, 如图 9-86 所示。



```

dhcp_agent_manager = neutron.agent.dhcp_agent.DhcpAgentwithStateReport
dhcp_driver = neutron.agent.linux.dhcp.Dnsmasq
interface_driver = neutron.agent.linux.interface.BridgeInterfaceDriver
OVS_use_veth = False
use_namespaces = True
debug = True
verbose = True

```

图 9-86

- `dhcp_driver`

使用 `dnsmasq` 实现 DHCP。

- `interface_driver`

使用 Linux Bridge 连接 DHCP namespace interface。

当创建 `network` 并在 `subnet` 上 `enable` DHCP 时，网络节点上的 DHCP agent 会启动一个 `dnsmasq` 进程为该 `network` 提供 DHCP 服务。

`dnsmasq` 是一个提供 DHCP 和 DNS 服务的开源软件。

`dnsmasq` 与 `network` 是一对一关系，一个 `dnsmasq` 进程可以为同一 `network` 中所有 `enable` 了 DHCP 的 `subnet` 提供服务。

回到我们的实验环境，之前创建了 `flat_net`，并且在 `subnet` 上启用了 DHCP，执行 `ps` 查看 `dnsmasq` 进程，如图 9-87 所示。

```

root@devstack-controller:~# ps -elf | grep dnsmasq | grep ns-19a0ed3d-fe
5 S nobody 4678 1 0 80 0 7053 poll_s 00:02 ? 00:00:00
0 dnsmasq: --no-hosts --no-resolv --strict-order --except-interface=lo --p
id-file=/opt/stack/data/neutron/dhcp/f153b42f-c3a1-4b6c-8865-c09b5b2aa274
pid --dhcp-hostsfile=/opt/stack/data/neutron/dhcp/f153b42f-c3a1-4b6c-886
5-c09b5b2aa274/host --addn-hosts=/opt/stack/data/neutron/dhcp/f153b42f-c3
a1-4b6c-8865-c09b5b2aa274/addn_hosts --dhcp-optsfile=/opt/stack/data/neut
ron/dhcp/f153b42f-c3a1-4b6c-8865-c09b5b2aa274/opts --dhcp-leasefile=/opt/
stack/data/neutron/dhcp/f153b42f-c3a1-4b6c-8865-c09b5b2aa274/leases --dch
p-match=set:ipxe,175 --bind-interfaces --interface=ns-19a0ed3d-fe --dhcp-
range=set:tag0,172.16.1.0,static,86400s --dhcp-lease-max=256 --conf-file=
--domain=openstacklocal
root@devstack-controller:~#

```

图 9-87

DHCP agent 会为每个 `network` 创建一个目录 `/opt/stack/data/neutron/dhcp/`，用于存放该 `network` 的 `dnsmasq` 配置文件。

下面讨论 `dnsmasq` 重要的启动参数：

- `--dhcp-hostsfile`

存放 DHCP host 信息的文件，这里的 `host` 在我们这里实际上就是 `instance`。

`dnsmasq` 从该文件获取 `host` 的 IP 与 MAC 的对应关系。

每个 `host` 对应一个条目，信息来源于 Neutron 数据库。

对于 `flat_net`，`hostsfile` 是 `/opt/stack/data/neutron/dhcp/f153b42f-c3a1-4b6c-8865-c09b5b2aa`

274/host, 记录了 DHCP, cirros-vm1 和 cirros-vm2 的 interface 信息, 如图 9-88 所示。

```
root@devstack-controller:~# cat /opt/stack/data/neutron/dhcp/f153b42f-c3a1-4b6c-8865-c09b5b2aa274/host
fa:16:3e:67:6b:f2,host-172-16-1-102.openstacklocal.,172.16.1.102
fa:16:3e:20:7f:5f,host-172-16-1-103.openstacklocal.,172.16.1.103
fa:16:3e:5e:37:de,host-172-16-1-104.openstacklocal.,172.16.1.104
root@devstack-controller:~#
```

图 9-88

### ● --interface

指定提供 DHCP 服务的 interface。dnsmasq 会在该 interface 上监听 instance 的 DHCP 请求。对于 flat\_net, interface 是 ns-19a0ed3d-fe。

或许大家还记得, 之前我们看到的 DHCP interface 叫 tap19a0ed3d-fe (如图 9-89 所示), 并非 ns-19a0ed3d-fe。

```
root@devstack-controller:~# brctl show
bridge name      bridge id        STP enabled      interfaces
brqf153b42f-c3  8000.005056b0b5ff  no               eth1
                                                         tap19a0ed3d-fe
                                                         tapc1875c7f-cb
```

图 9-89

从名称上看, ns-19a0ed3d-fe 和 tap19a0ed3d-fe 应该存在某种联系, 但那是什么呢?

要回答这个问题, 需要先讨论一个概念: Linux Network Namespace。

## 2. 用 Linux Network Namespace 隔离 dnsmasq 服务

在二层网络上, VLAN 可以将一个物理交换机分割成几个独立的虚拟交换机。类似地, 在三层网络上, Linux network namespace 可以将一个物理三层网络分割成几个独立的虚拟三层网络。每个 namespace 都有自己独立的网络栈, 包括 route table, firewall rule, network interface device 等。

Neutron 通过 namespace 为每个 network 提供独立的 DHCP 和路由服务, 从而允许租户创建重叠的网络。如果没有 namespace, 网络就不能重叠, 这样就失去了很多灵活性。

每个 dnsmasq 进程都位于独立的 namespace, 命名为 qdhcp-<network\_id>, 例如 flat\_net, 如图 9-90 所示。

```
root@devstack-controller:~# neutron net-list
+-----+-----+
| id                                     | name |
+-----+-----+
| f153b42f-c3a1-4b6c-8865-c09b5b2aa274 | flat_net |
+-----+-----+
root@devstack-controller:~# ip netns list
qdhcp-f153b42f-c3a1-4b6c-8865-c09b5b2aa274
root@devstack-controller:~#
```

图 9-90

ip netns list 命令列出所有的 namespace。

qdhcp-f153b42f-c3a1-4b6c-8865-c09b5b2aa274 就是 flat\_net 的 namespace。

其实，宿主机本身也有一个 namespace，叫 root namespace，拥有所有物理和虚拟 interface device。

物理 interface 只能位于 root namespace。

新创建的 namespace 默认只有一个 loopback device。

管理员可以将虚拟 interface，例如 bridge、tap 等设备添加到某个 namespace。

对于 flat\_net 的 DHCP 设备 tap19a0ed3d-fe，需要将其放到 namespace qdhcp-f153b42f-c3a1-4b6c-8865-c09b5b2aa274 中，但这样会带来一个问题：

tap19a0ed3d-fe 将无法直接与 root namespace 中的 bridge 设备 brqf153b42f-c3 连接。

Neutron 使用 veth pair 解决了这个问题。

veth pair 是一种成对出现的特殊网络设备，它们像一根虚拟的网线，可用于连接两个 namespace。向 veth pair 一端输入数据，在另一端就能读到此数据。

tap19a0ed3d-fe 与 ns-19a0ed3d-fe 就是一对 veth pair，它们将 qdhcp-f153b42f-c3a1-4b6c-8865-c09b5b2aa274 连接到 brqf153b42f-c3，如图 9-91 所示。

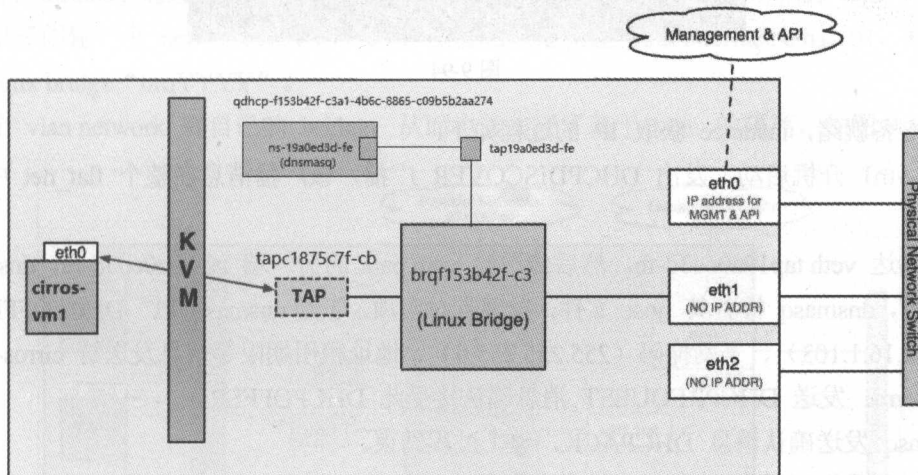


图 9-91

可以通过 `ip netns exec <network namespace name> <command>` 管理 namespace。

例如查看 ns-19a0ed3d-fe 的配置，如图 9-92 所示。

```
root@devstack-controller:~# ip netns exec qdhcp-f153b42f-c3a1-4b6c-8865-c09b5b2aa274 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: ns-19a0ed3d-fe: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether fa:16:3e:67:6b:f2 brd ff:ff:ff:ff:ff:ff
    inet 172.16.1.102/24 brd 172.16.1.255 scope global ns-19a0ed3d-fe
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe67:6bf2/64 scope link
        valid_lft forever preferred_lft forever
root@devstack-controller:~#
```

图 9-92

### 3. 分析 instance 如何从 dnsmasq 获取 IP

本节以 `cirros-vm1` 为例分析如何获取 DHCP IP。

在创建 instance 时, Neutron 会为其分配一个 port, 里面包含了 MAC 和 IP 地址信息。这些信息会同步更新到 dnsmasq 的 host 文件, 如图 9-93 所示。

```
root@devstack-controller:~# cat /opt/stack/data/neutron/dhcp/f153b42f-c3a1-4b6c-8865-c09b5b2aa274/host
fa:16:3e:67:3b:f2,host-172-16-1-102.openstacklocal,,172.16.1.102
fa:16:3e:20:7f:5f,host-172-16-1-103.openstacklocal,,172.16.1.103
fa:16:3e:5e:37:de,host-172-16-1-104.openstacklocal,,172.16.1.104
root@devstack-controller:~#
```



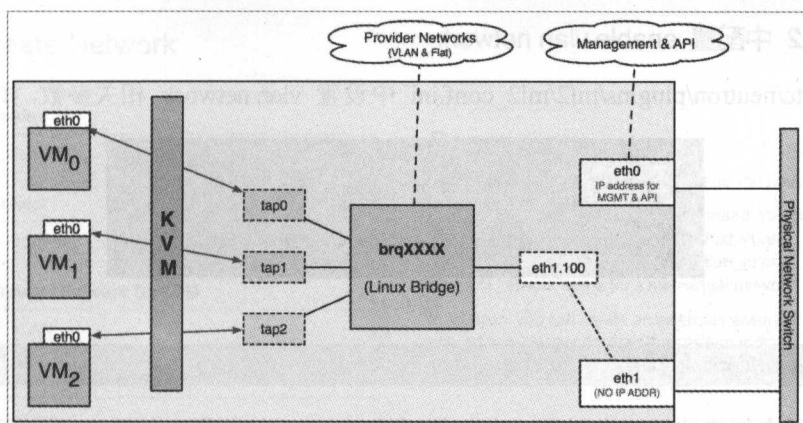


图 9-96

3 个 instance 通过 TAP 设备连接到名为“brqXXXX”linux bridge。

在物理网卡 eth1 上创建了 eth1.100 的 vlan interface, eth1.100 连接到 brqXXXX。

这样, instance 通过 eth1.100 发送到 eth1 的数据包就会打上 vlan100 的 tag。

如果多创建一个 network vlan101, eth1 上会相应地创建 vlan interface eth1.101, 并且连接的新的 linux bridge “brqYYYY”。

每个 vlan network 有自己的 bridge, 从而也就实现了基于 vlan 的隔离, 如图 9-97 所示。

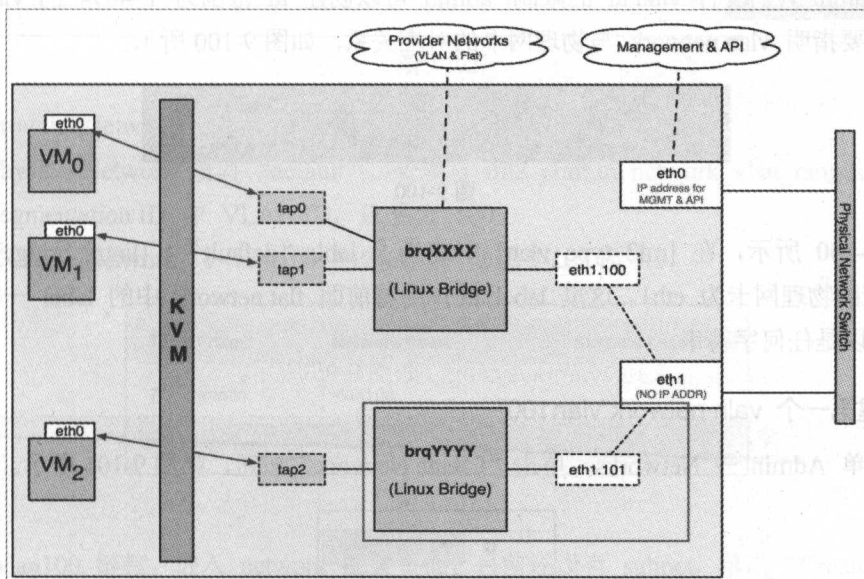


图 9-97

这里有一点要特别提醒:

因为物理网卡 eth1 上面可以走多个 vlan 的数据, 那么物理交换机上与 eth1 相连的 port 要设置成 trunk 模式, 而不是 access 模式。

接下来讨论如何在 Neutron 中配置 vlan 网络。

## 1. 在 ML2 中配置 enable vlan network

首先在 `/etc/neutron/plugins/ml2/ml2_conf.ini` 中设置 `vlan network` 相关参数, 如图 9-98 所示。

```
[ml2]
tenant_network_types = vlan
extension_drivers = port_security
type_drivers = local,flat,vlan,gre,vxlan
mechanism_drivers = linuxbridge
```

图 9-98

```
tenant_network_types = vlan
```

指定普通用户创建的网络类型为 `vlan`。然后指定 `vlan` 的范围, 如图 9-99 所示。

```
[ml2_type_vlan]
network_vlan_ranges = default:3001:4000
```

图 9-99

上面配置定义了 `label` 为 “`default`” 的 `vlan network`, `vlan id` 的范围是 3001~4000。这个范围是针对普通用户在自己的租户里创建 `network` 的范围。因为普通用户创建 `network` 时并不能指定 `vlan id`, Neutron 会按顺序自动从这个范围中取值。

对于 `admin`, 则没有 `vlan id` 的限制, `admin` 可以创建 `id` 范围为 1~4094 的 `vlan network`。接着需要指明 `vlan network` 与物理网卡的对应关系, 如图 9-100 所示。

```
[linux_bridge]
physical_interface_mappings = default:eth1
```

图 9-100

如图 9-100 所示, 在 `[ml2_type_vlan]` 中定义了 `lable` “`default`”, `[linux_bridge]` 中则指明 `default` 对应的物理网卡为 `eth1`。这里 `label` 的作用与前面 `flat network` 中的 `label` 一样, 只是一个标示, 可以是任何字符串。

## 2. 创建第一个 valn network“vlan100”

打开菜单 `Admin → Networks`, 单击 “`Create Network`” 按钮, 如图 9-101 所示。

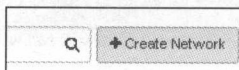
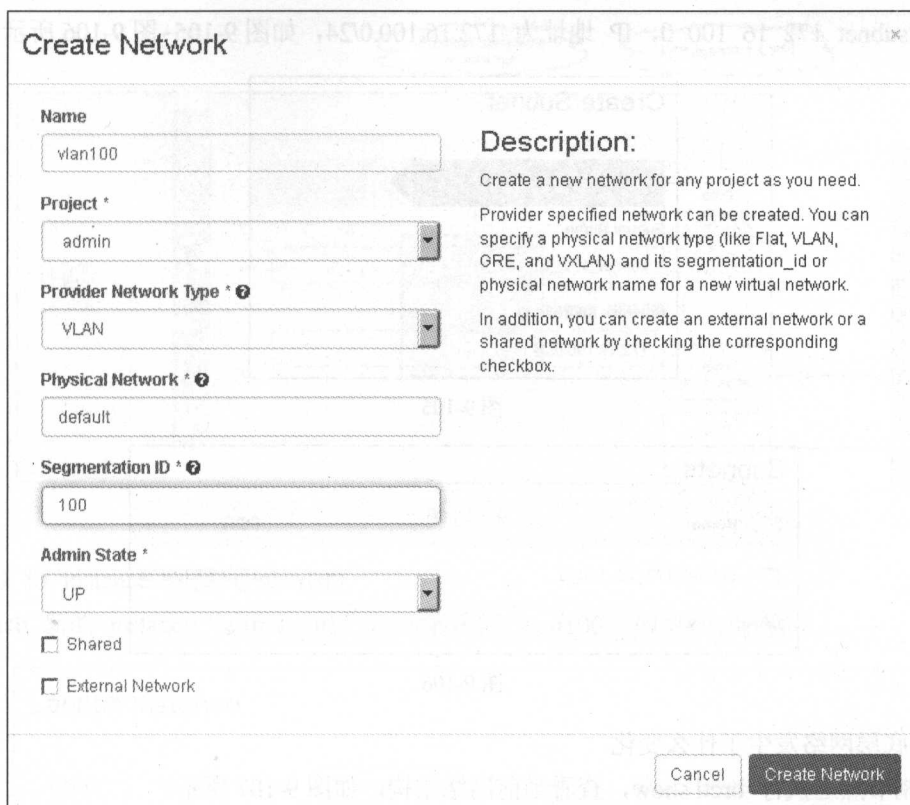


图 9-101

显示创建页面, 如图 9-102 所示。



**Create Network**

**Name**  
vlan100

**Project \***  
admin

**Provider Network Type \* ?**  
VLAN

**Physical Network \* ?**  
default

**Segmentation ID \* ?**  
100

**Admin State \***  
UP

☐ Shared

☐ External Network

**Description:**  
Create a new network for any project as you need.  
Provider specified network can be created. You can specify a physical network type (like Flat, VLAN, GRE, and VXLAN) and its segmentation\_id or physical network name for a new virtual network.  
In addition, you can create an external network or a shared network by checking the corresponding checkbox.

Cancel Create Network

图 9-102

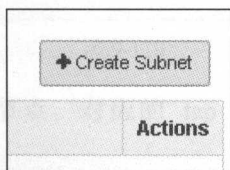
- Provider Network Type 选择“VLAN”。
- Physical Network 填写“default”，必须与 ml2\_conf.ini network\_vlan\_ranges 保持一致。
- Segmentation ID 即 VLAN ID，设置为 100。

单击“Create Network”，vlan100 创建成功，如图 9-103 所示。

<input type="checkbox"/>	Project	Network Name	Subnets Associated
<input type="checkbox"/>	admin	vlan100	
Displaying 1 item			

图 9-103

单击 vlan100 链接，进入 network 配置页面，目前还没有 subnet，单击“Create Subnet”按钮，如图 9-104 所示。



+ Create Subnet

Actions

图 9-104

创建 `subnet_172_16_100_0`, IP 地址为 `172.16.100.0/24`, 如图 9-105~图 9-106 所示。

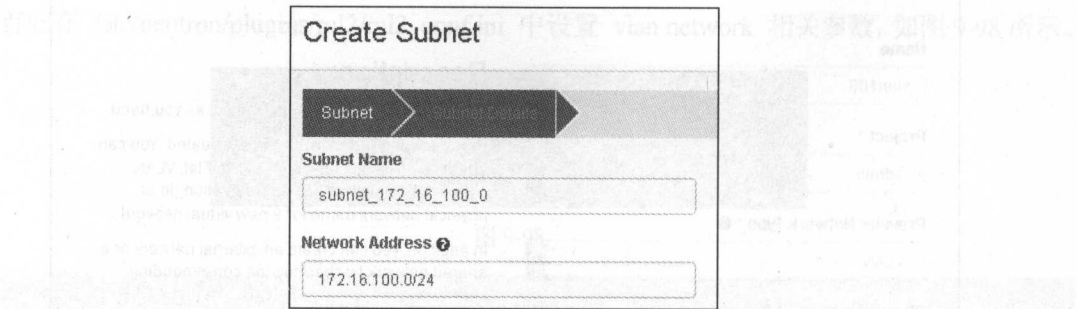


图 9-105

Subnets		
<input type="checkbox"/>	Name	CIDR
<input type="checkbox"/>	subnet_172_16_100_0	172.16.100.0/24
Displaying 1 item		

图 9-106

(1) 底层网络发生了什么变化

在控制节点上执行 `brctl show`, 查看当前网络结构, 如图 9-107 所示。

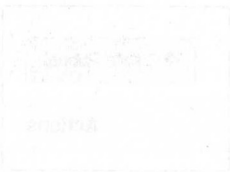
```
root@devstack-controller:~# brctl show
bridge name      bridge id        STP enabled      interfaces
brq3fcfdb98-9d    8000.005056b0b5ff  no               eth1.100
                  tap1180bbe8-06
virbr0            8000.000000000000  yes
```

图 9-107

Neutron 自动新建了三个设备:

- `vlan100` 对应的网桥为 `brq3fcfdb98-9d`。
- `vlan interface` 为 `eth1.100`。
- `dhcp` 的 `tap` 设备为 `tap1180bbe8-06`。

`eth1.100` 和 `tap19a0ed3d-fe` 已经连接到了 `brq3fcfdb98-9d`, `VLAN 100` 的二层网络就绪, 此时 `vlan100` 结构如图 9-108 所示。





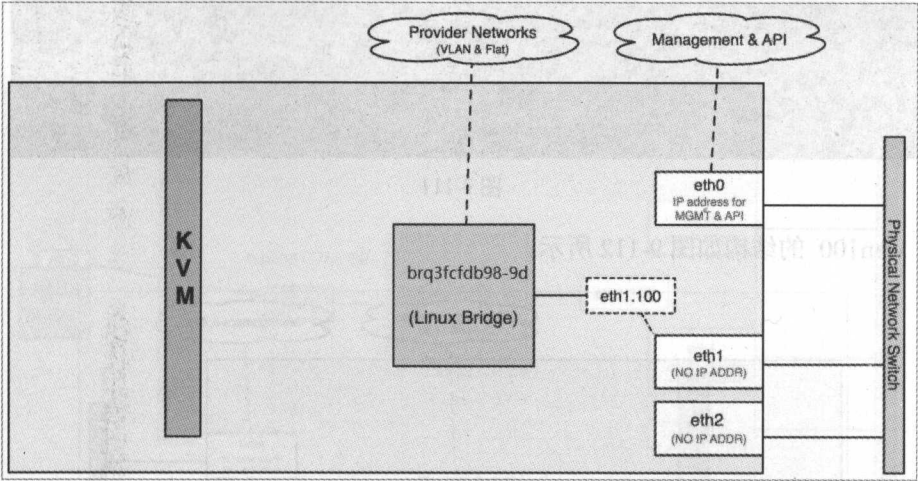


图 9-108

(2) 将 instance 连接到 vlan100  
launch 新的 instance “cirros-vm1”，网络选择 vlan100，如图 9-109 所示。

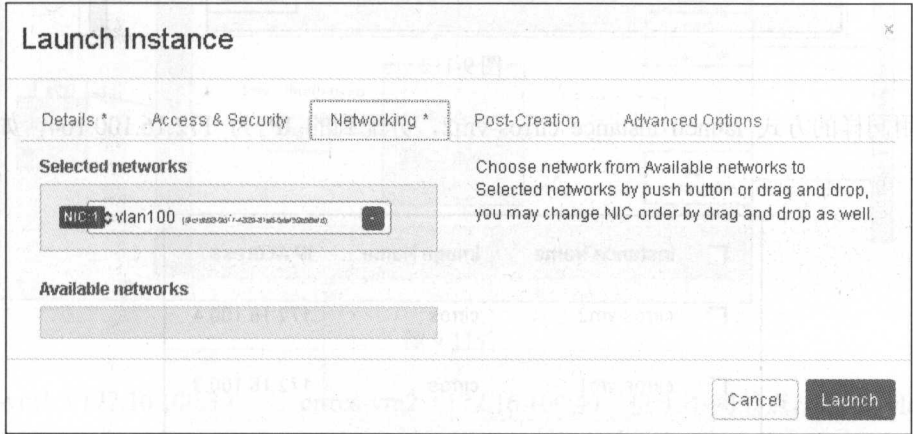


图 9-109

cirros-vm1 分配到的 IP 为 172.16.100.3，如图 9-110 所示。

<input type="checkbox"/>	Instance Name	Image Name	IP Address
<input type="checkbox"/>	cirros-vm1	cirros	172.16.100.3
Displaying 1 item			

图 9-110

cirros-vm1 被 schedule 到控制节点，对应的 tap 设备为 tap238437b8-50，并且连接到 bridge，如图 9-111 所示。

```
root@devstack-controller:~# brctl show
bridge name      bridge id        STP enabled      interfaces
brq3fcfdb98-9d   8000.005056b0b5ff  no               eth1.100
                                                         tap1180bbe8-06
                                                         tap238437b8-50
virbr0           8000.000000000000  yes
root@devstack-controller:~#
```

图 9-111

当前 vlan100 的结构如图 9-112 所示。

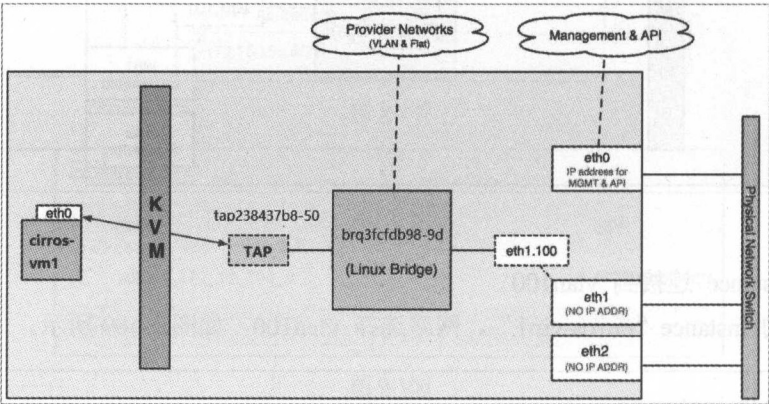


图 9-112

继续用同样的方式 launch instance cirros-vm2，分配到的 IP 为 172.16.100.104，如图 9-113 所示。

<input type="checkbox"/>	Instance Name	Image Name	IP Address
<input type="checkbox"/>	cirros-vm2	cirros	172.16.100.4
<input type="checkbox"/>	cirros-vm1	cirros	172.16.100.3
Displaying 2 items			

图 9-113

cirros-vm2 被 schedule 到计算节点，对应的 tap 设备为 tapac94e0e8-2b，并且连接到 bridge，如图 9-114 所示。

```
root@devstack-compute1:~# brctl show
bridge name      bridge id        STP enabled      interfaces
brq3fcfdb98-9d   8000.005056b0b540  no               eth1.100
                                                         tapac94e0e8-2b
virbr0           8000.000000000000  yes
root@devstack-compute1:~#
```

图 9-114

因为计算节点上没有 HDCP 服务，所以没有相应的 tap 设备。  
另外，bridge 的名称与控制节点上一致，都是 brq3fcfdb98-9d，表明是同一个 network。

当前 vlan100 的结构如图 9-115 所示。

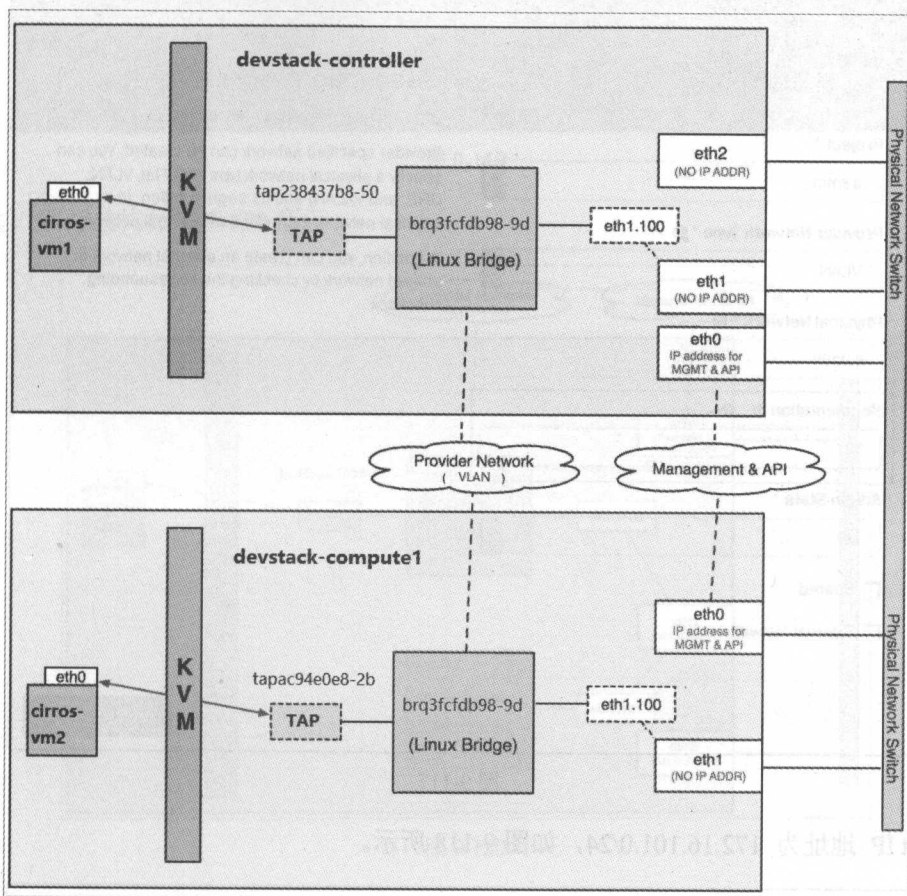


图 9-115

cirros-vm1 (172.16.100.3) 与 cirros-vm2 (172.16.100.4) 位于不同节点, 通过 vlan100 相连, 下面执行 PING 验证连通性。

在 cirros-vm1 控制台中执行 ping 172.16.100.4, 如图 9-116 所示。

```

$ ping 172.16.100.4
PING 172.16.100.4 (172.16.100.4): 56 data bytes
64 bytes from 172.16.100.4: seq=0 ttl=64 time=2.603 ms
64 bytes from 172.16.100.4: seq=1 ttl=64 time=1.730 ms
64 bytes from 172.16.100.4: seq=2 ttl=64 time=1.884 ms
64 bytes from 172.16.100.4: seq=3 ttl=64 time=1.751 ms
64 bytes from 172.16.100.4: seq=4 ttl=64 time=1.912 ms
64 bytes from 172.16.100.4: seq=5 ttl=64 time=1.977 ms
64 bytes from 172.16.100.4: seq=6 ttl=64 time=1.537 ms

```

图 9-116

如我们预料, ping 成功。

### 3. 创建第二个 valn network “vlan101”

继续创建第二个 vlan network “vlan101”, 如图 9-117 所示。

Create Network

Name

vlan101

Project \*

admin

Provider Network Type \* ?

VLAN

Physical Network \* ?

default

Segmentation ID \* ?

101

Admin State \*

UP

☐ Shared

☐ External Network

Description:

Create a new network for any project as you need.  
Provider specified network can be created. You can specify a physical network type (like Flat, VLAN, GRE, and VXLAN) and its segmentation\_id or physical network name for a new virtual network.  
In addition, you can create an external network or a shared network by checking the corresponding checkbox.

Cancel

Create Network

图 9-117

subnet IP 地址为 172.16.101.0/24，如图 9-118 所示。

<input type="checkbox"/>	Project	Network Name	Subnets Associated
<input type="checkbox"/>	admin	vlan101	subnet_172_16_101_0 172.16.101.0/24
<input type="checkbox"/>	admin	vlan100	subnet_172_16_100_0 172.16.100.0/24
Displaying 2 items			

图 9-118

(1) 底层网络发生了什么变化

Neutron 自动创建了 vlan101 对应的网桥 brq1d7040b8-01，vlan interface eth1.101，以及 dhcp 的 tap 设备 tap5b1a2247-32。

eth1.101 和 tap5b1a2247-32 已经连接到 brq1d7040b8-01，VLAN 101 的二层网络就绪，如图 9-119 所示。



```

root@devstack-controller:~# brctl show
bridge name      bridge id        STP enabled      interfaces
brq1d7040b8-01   8000.005056b0b5ff  no               eth1.101
                                                           tap5b1a2247-32
brq3fcfdb98-9d   8000.005056b0b5ff  no               eth1.100
                                                           tap1180bbe8-06
                                                           tap238437b8-50
virbr0           8000.000000000000  yes
root@devstack-controller:~#

```

图 9-119

现在，网络结构如图 9-120 所示。

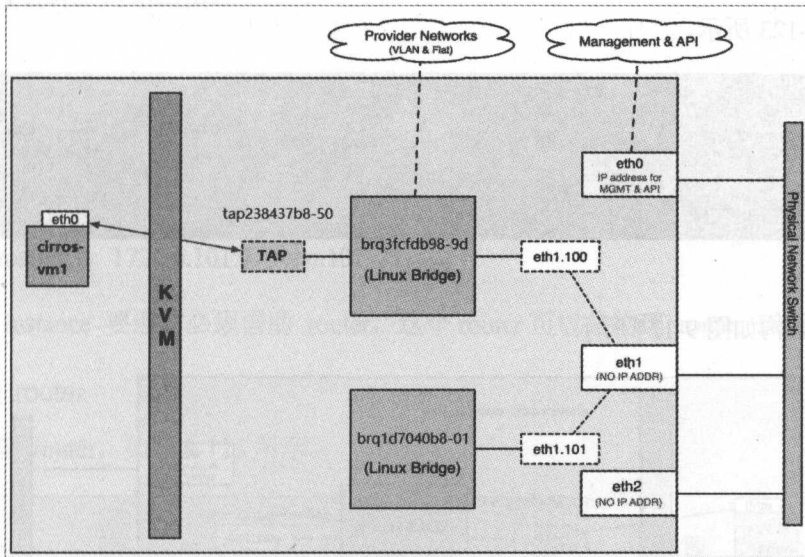


图 9-120

(2) 将 instance 连接到 vlan101

launch 新的 instance “cirros-vm3”，网络选择 vlan101，如图 9-121 所示。

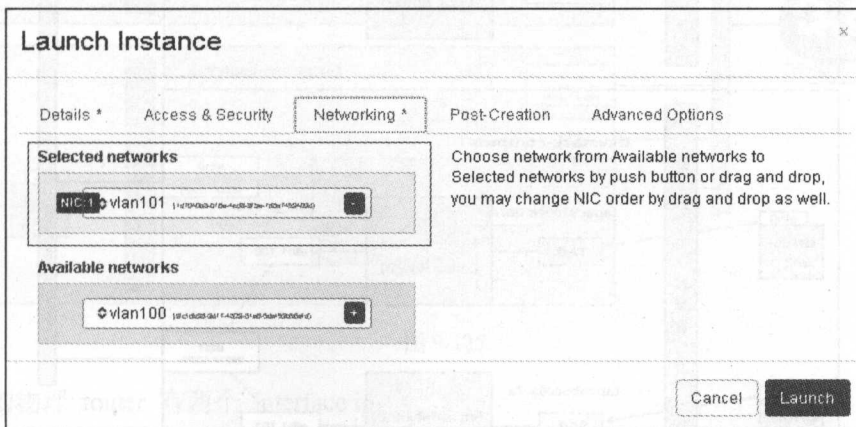


图 9-121

cirros-vm3 分配到的 IP 为 172.16.101.103，如图 9-122 所示。

<input type="checkbox"/>	Instance Name	Image Name	IP Address
<input type="checkbox"/>	cirros-vm3	cirros	172.16.101.3
<input type="checkbox"/>	cirros-vm2	cirros	172.16.100.4
<input type="checkbox"/>	cirros-vm1	cirros	172.16.100.3

Displaying 3 Items

图 9-122

cirros-vm3 被 schedule 到计算节点，对应的 tap 设备为 tapadb5cc6a-7a，并且连接到 bridge，如图 9-123 所示。

```
root@devstack-compute1:~# brctl show
bridge name      bridge id        STP enabled      interfaces
brq1d7040b8-01   8000.005056b0b540  no               eth1.101
                                                           tapadb5cc6a-7a
brq3fcfdb98-9d   8000.005056b0b540  no               eth1.100
                                                           tapac94e0e8-2b
virbr0           8000.000000000000  yes
```

图 9-123

当前网络结构如图 9-124 所示。

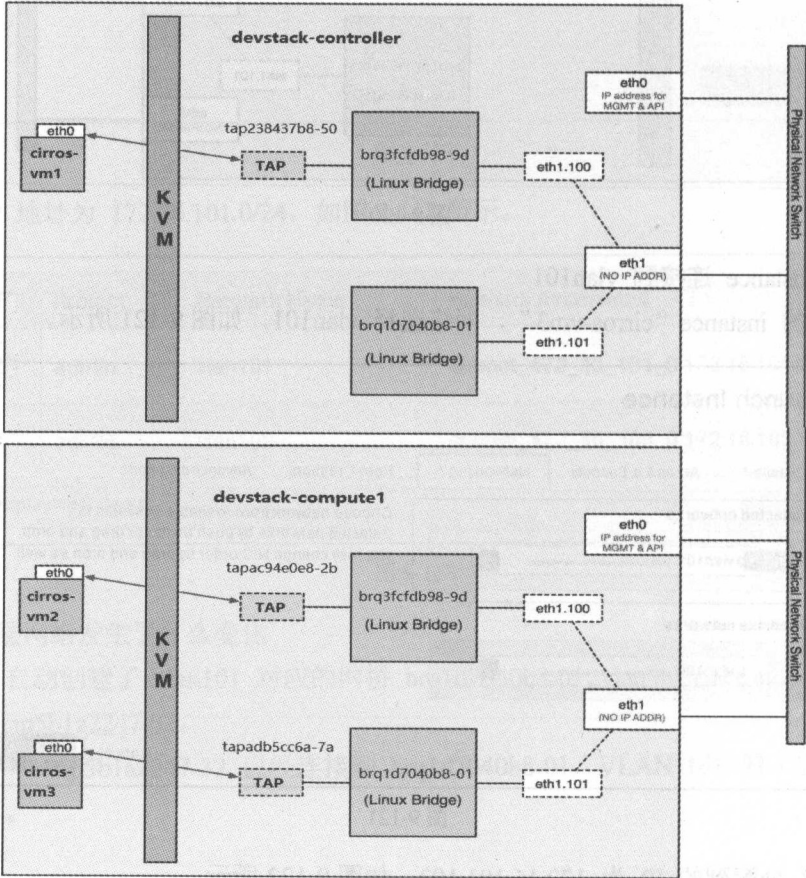


图 9-124

- cirros-vm1 位于控制节点，属于 vlan100。
- cirros-vm2 位于计算节点，属于 vlan100。
- cirros-vm3 位于计算节点，属于 vlan101。

cirros-vm1 与 cirros-vm2 都在 vlan100，它们之间能通信。

cirros-vm3 在 vlan101，不能与 cirros-vm1 和 cirros-vm2 通信。那怎么样才能让 vlan100 与 vlan101 中的 instance 通信呢？二层 vlan 是不行的，只能在三层通过路由器转发。

下一节便会详细讨论路由。

## 9.4.8 Routing

路由服务提供跨 subnet 互联互通功能。例如，前面我们搭建了实验环境：

- cirros-vm1, 172.16.100.3, vlan100
- cirros-vm3, 172.16.101.3, vlan101

这两个 instance 要通信必须借助 router，这个 router 可以是物理 router 或者虚拟 router。

### 1. 物理 router

使用物理 router，如图 9-125 所示。

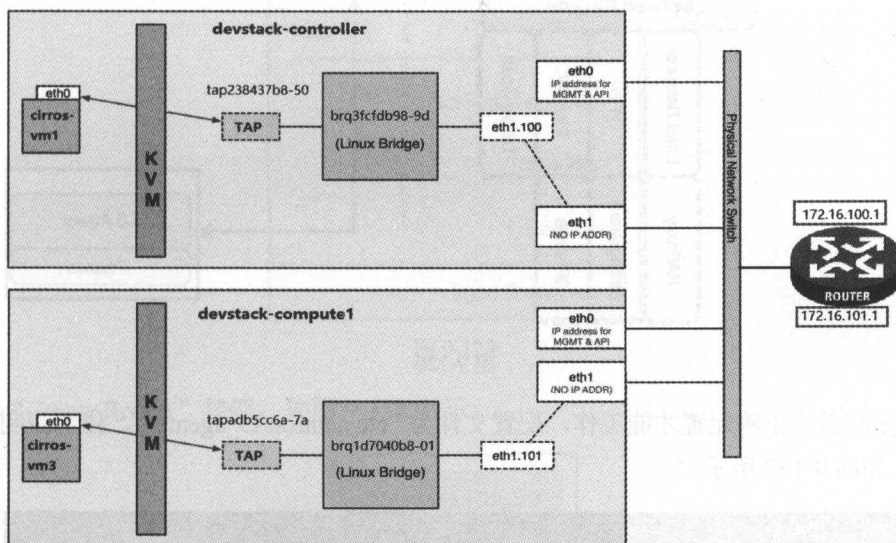


图 9-125

接入的物理 router 有两个 interface ip:

- 172.16.100.1 对应 vlan100 的网关。
- 172.16.101.1 对应 vlan101 的网关。

当 cirros-vm1 要跟 cirros-vm3 通信时，数据包的流向是这样的：

- (1) 因为 cirros-vm1 的默认网关指向 172.16.100.1，cirros-vm1 发送到 cirros-vm3 的数据包首先通过 vlan100 的 interface 进入物理 router。
- (2) router 发现目的地址是 172.16.101.1，则从 vlan101 的 interface 发出。
- (3) 数据包经过 brq1d7040b8-01 最终到达 cirros-vm3。

2. 虚拟 router

虚拟 router 的路由机制与物理 router 一样，只是由软件实现。

Neutron 两种方案都支持。如果要使用虚拟 router，L3 Agent 会在控制节点或者网络节点上运行虚拟 router，为 subnet 提供路由服务。

下面详细讨论 Neutron 的虚拟 router 实现。

(1) 配置 L3 Agent

Neutron 的路由服务是由 L3 Agent 提供的。除此之外，L3 Agent 通过 iptables 提供 firewall 和 floating ip 服务，如图 9-126 所示。

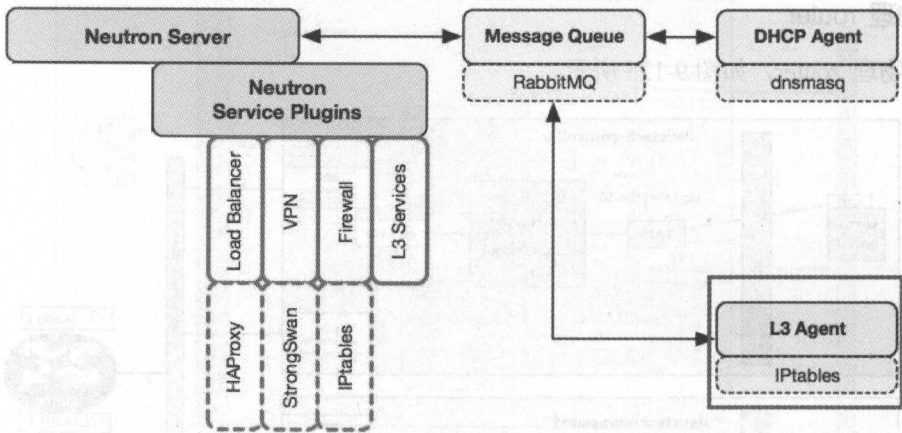


图 9-126

L3 Agent 需要正确配置才能工作，配置文件为 /etc/neutron/l3\_agent.ini，位于控制节点或网络节点上，如图 9-127 所示。

```
[DEFAULT]  
interface_driver = neutron.agent.linux.interface.BridgeInterfaceDriver
```

图 9-127

interface\_driver 是最重要的选项，如果 mechanism driver 是 linux bridge，则：

```
interface_driver = neutron.agent.linux.interface.BridgeInterfaceDriver
```

如果选用 open vswitch，则：



```
interface_driver = neutron.agent.linux.interface.OVSInterfaceDriver
```

L3 Agent 运行在控制或网络节点上，如图 9-128 所示。

```
root@devstack-controller:~# source /opt/stack/devstack/openrc admin admin
root@devstack-controller:~# neutron agent-list
```

id	agent_type	host	alive
0b708280-6724-4f3f-89f3-241793516d33	DHCP agent	devstack-controller	--
4f92dc19-a298-4fae-86e4-40d7fc64153d	Linux bridge agent	devstack-controller	--
992f214f-e537-4c92-8c91-b7f54cfd1ab	Loadbalancer agent	devstack-controller	--
a1b26085-5547-483a-810e-008a95529f24	L3 agent	devstack-controller	--
d2695f39-a99e-4f81-a20a-2ed545874558	Linux bridge agent	devstack-compute1	--
f1a7924c-1e12-4643-a1d0-f58a8e30119b	Metadata agent	devstack-controller	--

图 9-128

## (2) 用虚拟 router 实现 subnet 间路由

下面将创建虚拟 router “router\_100\_101”，打通 vlan100 和 vlan101。

首先创建 router。操作菜单 Project → Network → Routers，如图 9-129 所示。

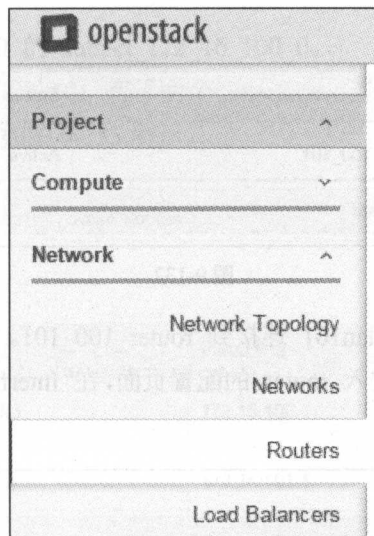


图 9-129

单击“Create Router”按钮，如图 9-130 所示。

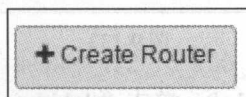


图 9-130

router 命名为“router\_100\_101”，单击“Create Router”确认，如图 9-131 所示。

Create Router

Router Name \*

router\_100\_101

Admin State

UP

Description:

Creates a router with specified parameters.

Cancel

Create Router

图 9-131

router\_100\_101 创建成功，如图 9-132 所示。

Routers

<input type="checkbox"/>	Name	Status
<input type="checkbox"/>	router_100_101	Active
Displaying 1 item		

图 9-132

接下来需要将 vlan100 和 vlan101 连接到 router\_100\_101。

单击“router\_100\_101”链接进入 router 的配置页面，在“Interfaces”标签中单击“Add Interface”按钮，如图 9-133 所示。

Overview

Interfaces

Static Routes

+ Add Interface

Name	Fixed IPs	Status	Type	Admin State	Actions
No items to display.					
Displaying 0 items					

图 9-133

选择 vlan101 的 subnet\_172\_16\_101\_0，单击 “Add Interface” 确认，如图 9-134 所示。

**Add Interface**

**Subnet \***  
 vlan101: 172.16.101.0/24 (subnet\_172\_16\_101)

**IP Address (optional) ⓘ**

**Router Name \***  
 router\_100\_101

**Router ID \***  
 6c04fc3f-53e2-4eea-9cb7-f31564649c78

**Description:**  
 You can connect a specified subnet to the router.  
 The default IP address of the interface created is a gateway of the selected subnet. You can specify another IP address of the interface here. You must select a subnet to which the specified IP address belongs to from the above list.

Cancel Add interface

图 9-134

用同样的方法添加 vlan100 的 subnet\_172\_16\_100\_0。

完成后, 可以看到 router\_100\_101 有了两个 interface, 其 IP 正好是 subnet 的 Gateway IP 172.16.100.1 和 172.16.101.1, 如图 9-135 所示。

<input type="checkbox"/>	Name	Fixed IPs	Status
<input type="checkbox"/>	(d568ba1a-740e)	172.16.100.1	Active
<input type="checkbox"/>	(e17162c5-00fa)	172.16.101.1	Active

Displaying 2 items

图 9-135

到这里, 我们可以预见:

- router\_100\_101 已经连接了 subnet\_172\_16\_100\_0 和 subnet\_172\_16\_101\_0。
- router\_100\_101 上已经设置好了两个 subnet 的 Gateway IP。
- cirros-vm1 和 cirros-vm3 应该可以通信了。

通过 PING 测试一下, 如图 9-136 所示。

```
$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        inet6 ::1/128 scope host
            valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
    link/ether fa:16:3e:c4:9a:29 brd ff:ff:ff:ff:ff:ff
    inet 172.16.100.3/24 brd 172.16.100.255 scope global
        inet6 fe80::f816:3eff:fec4:9a29/64 scope link
            valid_lft forever preferred_lft forever
$ ping 172.16.101.3
PING 172.16.101.3 (172.16.101.3): 56 data bytes
64 bytes from 172.16.101.3: seq=0 ttl=63 time=4.310 ms
64 bytes from 172.16.101.3: seq=1 ttl=63 time=2.402 ms
64 bytes from 172.16.101.3: seq=2 ttl=63 time=1.558 ms
64 bytes from 172.16.101.3: seq=3 ttl=63 time=2.242 ms
64 bytes from 172.16.101.3: seq=4 ttl=63 time=1.976 ms
64 bytes from 172.16.101.3: seq=5 ttl=63 time=2.052 ms
```

图 9-136

判断正确，cirros-vm1 和 cirros-vm3 能通信了。

查看 cirros-vm1 的路由表，默认网关为 172.16.100.1。

同时 traceroute 告诉我们，cirros-vm1 确实是通过 router\_100\_101 访问到 cirros-vm3 的，如图 9-137 所示。

```
$ route
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
default 172.16.100.1 0.0.0.0 UG 0 0 0 eth0
172.16.100.0 * 255.255.255.0 U 0 0 0 eth0
$ traceroute 172.16.101.3
traceroute to 172.16.101.3 (172.16.101.3), 30 hops max, 46 byte packets
 1 172.16.100.1 (172.16.100.1) 0.201 ms 0.370 ms 0.505 ms
 2 172.16.101.3 (172.16.101.3) 1.911 ms 1.512 ms 1.050 ms
```

图 9-137

下面研究 router\_100\_101 是如何起作用的。

### (3) 底层网络发生了什么变化

查看控制节点的 Linux Bridge 结构发生了什么变化，如图 9-138 所示。

```
root@devstack-controller:~# brctl show
bridge name bridge id STP enabled interfaces
brq1d7040b8-01 8000.005056b0b5ff no eth1.101
tap5b1a2247-32
tapel7162c5-00
brq3fcfdb98-9d 8000.005056b0b5ff no eth1.100
tap1180bbe8-06
tap238437b8-50
tapd568bala-74
virbr0 8000.000000000000 yes
```

图 9-138

vlan101 的 bridge 上多了一个 tapel7162c5-00，从命名上可以推断该 TAP 设备对应 router\_100\_101 的 interface (e17162c5-00fa)。



vlan100 的 bridge 上多了一个 tapd568bala-74，从命名上可以推断该 TAP 设备对应 router\_100\_101 的 interface (d568bala-740e)。

当前网络结构如图 9-139 所示。

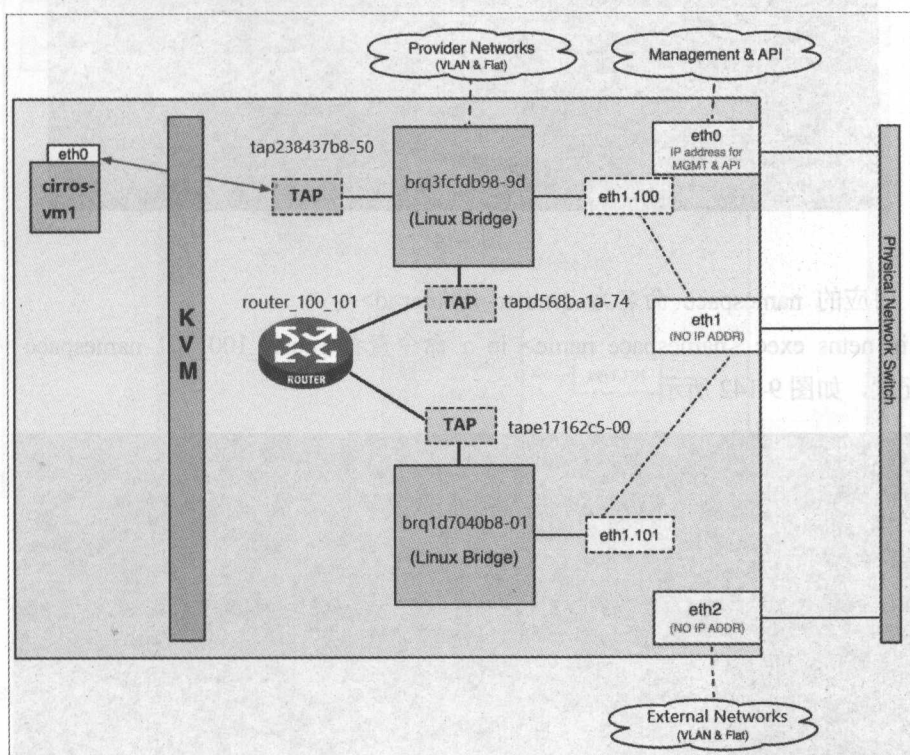


图 9-139

但发现一个问题：两个 TAP 设备上并没有配置相应的 Gateway IP，如图 9-140 所示。

```
root@devstack-controller:~# ifconfig tape17162c5-00
tape17162c5-00 Link encap:Ethernet HWaddr a6:6e:0d:a2:79:5a
  inet6 addr: fe80::a46e:dff:fea2:795a/64 Scope:Link
    UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
    RX packets:120 errors:0 dropped:0 overruns:0 frame:0
    TX packets:118 errors:0 dropped:0 overruns:0 carrier:0
    collisions:0 txqueuelen:1000
    RX bytes:10856 (10.8 KB) TX bytes:10988 (10.9 KB)

root@devstack-controller:~# ifconfig tapd568bala-74
tapd568bala-74 Link encap:Ethernet HWaddr 26:ce:5c:8c:53:82
  inet6 addr: fe80::24ce:5cff:fe8c:5382/64 Scope:Link
    UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
    RX packets:126 errors:0 dropped:0 overruns:0 frame:0
    TX packets:1016 errors:0 dropped:0 overruns:0 carrier:0
    collisions:0 txqueuelen:1000
    RX bytes:11552 (11.5 KB) TX bytes:93132 (93.1 KB)
```

图 9-140

如果没有 Gateway IP，router\_100\_101 是如何完成路由的呢？

答案是：L3 Agent 会为每个 router 创建了一个 namespace，通过 veth pair 与 TAP 相连，然后将 Gateway IP 配置在位于 namespace 里面的 veth interface 上，这样就能提供路由了。

通过 ip netns 查看 namespace，如图 9-141 所示。

```
root@devstack-controller:~# neutron router-list
+-----+-----+
| id                                           | name |
+-----+-----+
| 6c04fc3f-53e2-4eea-9cb7-f31564649c78 | router_100_101 |
+-----+-----+
root@devstack-controller:~#
root@devstack-controller:~# ip netns
qrouter-6c04fc3f-53e2-4eea-9cb7-f31564649c78
qdhcp-3fcfdb98-9d17-4829-81e6-5def59b56efd
qdhcp-1d7040b8-016e-4cd8-9f5e-7d8e7453403d
root@devstack-controller:~#
```

图 9-141

router 对应的 namespace 命名为 qrouter-**<router id>**。

通过 ip netns exec <namespace name> ip a 命令查看 router\_100\_101 namespace 中的 veth interface 配置，如图 9-142 所示。

```
root@devstack-controller:~# ip netns exec qrouter-6c04fc3f-53e2-4eea-9cb7-
-f31564649c78 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: qr-el7162c5-00: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo
_fast state UP group default qlen 1000
    link/ether fa:16:3e:89:15:ba brd ff:ff:ff:ff:ff:ff
    inet 172.16.101.1/24 brd 172.16.101.255 scope global qr-el7162c5-00
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe89:15ba/64 scope link
        valid_lft forever preferred_lft forever
3: qr-d568bala-74: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo
_fast state UP group default qlen 1000
    link/ether fa:16:3e:07:92:a5 brd ff:ff:ff:ff:ff:ff
    inet 172.16.100.1/24 brd 172.16.100.255 scope global qr-d568bala-74
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe07:92a5/64 scope link
        valid_lft forever preferred_lft forever
root@devstack-controller:~#
```

图 9-142

namespace 中有两个 interface:

- qr-el7162c5-00 上设置了 Gateway IP 172.16.101.1，与 root namespace 中的 tapel7162c5-00 组成 veth pair。
- qr-d568bala-74 上设置了 Gateway IP 172.16.100.1，与 root namespace 中的 tapd568bala-74 组成 veth pair。

网络结构如图 9-143 所示。

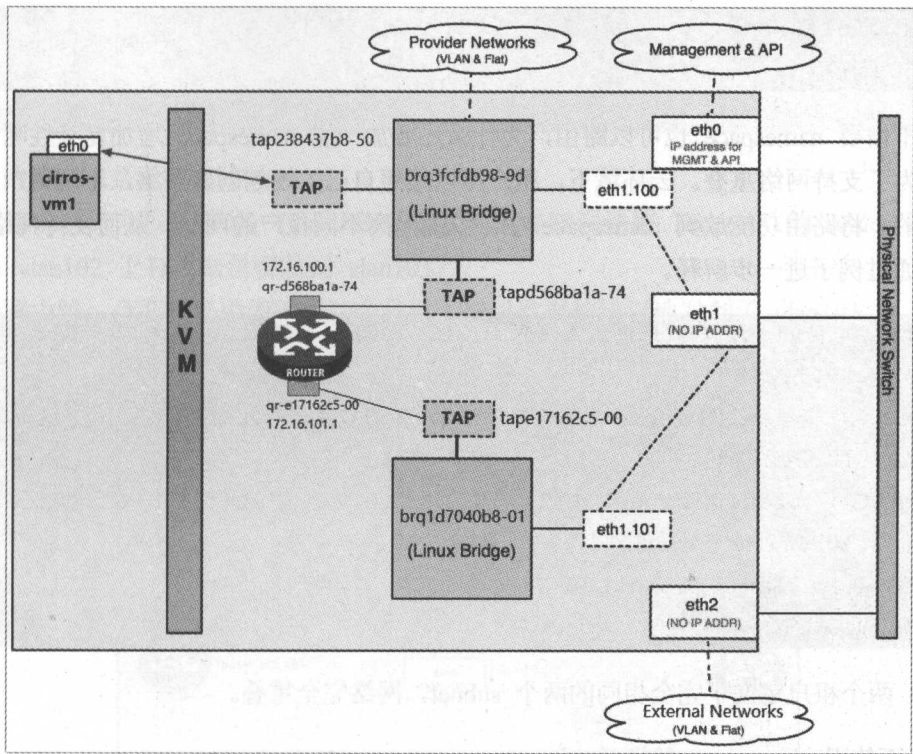


图 9-143

namespace 中的路由表也保证了 subnet\_172\_16\_100\_0 和 subnet\_172\_16\_101\_0 之间是可以路由的，如图 9-144 所示。

```
root@devstack-controller:~# ip netns exec qrouter-6c04fc3f-53e2-4eea-9cb7-f31564649c78
kernel IP routing table
Destination      Gateway         Genmask         Flags Metric Ref    Use Iface
172.16.100.0     *              255.255.255.0   U        0      0      0 qr-d568ba1a-74
172.16.101.0     *              255.255.255.0   U        0      0      0 qr-e17162c5-00
root@devstack-controller:~#
```

图 9-144

(4) 为什么要使用 namespace

接下来我们讨论一个更深层次的问题：

为什么不直接在 tape17162c5-00 和 tapd568ba1a-74 上配置 Gateway IP，而是引入一个 namespace，在 namespace 里面配置 Gateway IP 呢？

首先考虑另外一个问题：

如果不用 namespace，直接 Gateway IP 配置到 tape17162c5-00 和 tapd568ba1a-74 上，能不能连通 subnet\_172\_16\_100\_0 和 subnet\_172\_16\_101\_0 呢？

答案是可以的，只要控制节点上配置了类似下面的路由。

```
Destination      Gateway         Genmask         Flags Metric Ref    Use Iface
172.16.100.0     *              255.255.255.0   U        0      0      0
tapd568ba1a-74
```



```
172.16.101.0 * 255.255.255.0 U 0 0 0
tap17162c5-00
```

既然不需要 namespace 也可以路由，为什么还要加一层 namespace 增加复杂性呢？其根本原因是：为了支持网络重叠。云环境下，租户可以按照自己的规划创建网络，不同租户的网络是可能重叠的。将路由功能放到 namespace 中，就能隔离不同租户的网络，从而支持网络重叠。下面通过例子进一步解释。

```
Tenant A   vlan100subnet A-1: 10.10.1.0/24 {"start": "10.10.1.1",
"end": "10.10.1.254"}
Tenant A   vlan101subnet A-2: 10.10.2.0/24 {"start": "10.10.2.1",
"end": "10.10.2.254"}
Tenant B   vlan102subnet B-1: 10.10.1.0/24 {"start": "10.10.1.1",
"end": "10.10.1.254"}
Tenant B   vlan103subnet B-2: 10.10.2.0/24 {"start": "10.10.2.1",
"end": "10.10.2.254"}
```

A、B 两个租户定义了完全相同的两个 subnet，网络完全重叠。

(5) 不使用 namespace 的场景

如果不使用 namespace，网络结构如图 9-145 所示。

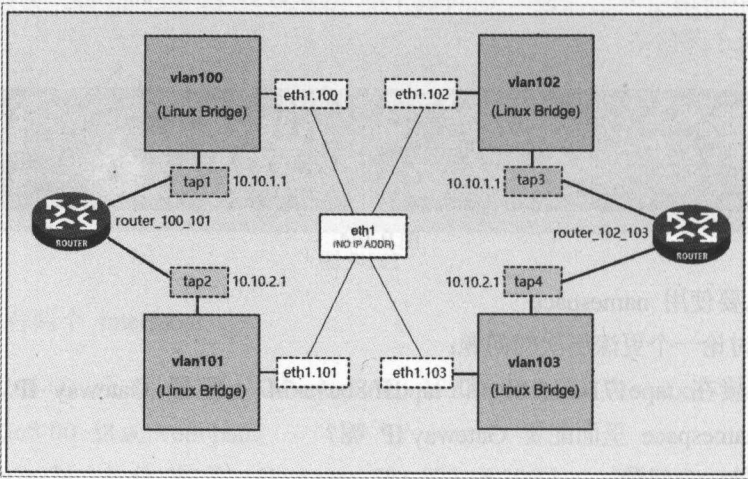


图 9-145

其特征是网关 IP 配置在 TAP interface 上。

因为没有 namespace 隔离，router\_100\_101 和 router\_102\_103 的路由条目都只能记录到控制节点操作系统（root namespace）的路由表中，内容如下：

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
10.10.1.0	*	255.255.255.0	U	0	0	0	tap1



10.10.2.0	*	255.255.255.0	U	0	0	0	tap2
10.10.1.0	*	255.255.255.0	U	0	0	0	tap3
10.10.2.0	*	255.255.255.0	U	0	0	0	tap4

这样的路由表是无法工作的。

按照路由表优先匹配原则，Tenant B 的数据包总是错误地被 Tenant A 的 router 路由。

例如 vlan102 上有数据包要发到 vlan103。

选择路由时，会匹配路由表的第二个条目，结果数据被错误地发到了 vlan101。

#### (6) 使用 namespace 的场景

如果使用 namespace，网络结构如图 9-146 所示。

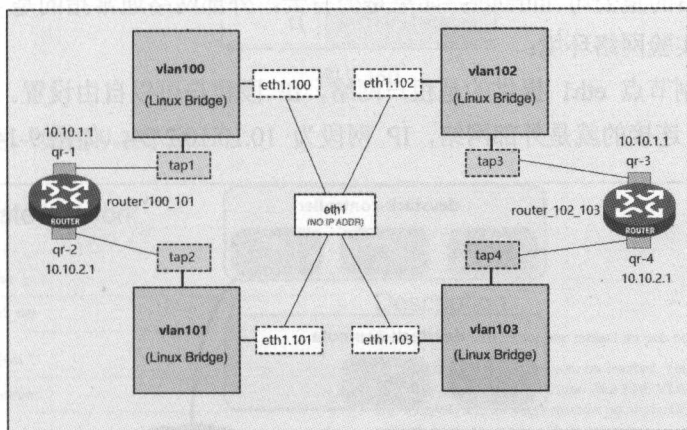


图 9-146

其特征是网关 IP 配置在 namespace 中的 veth interface 上。

每个 namespace 拥有自己的路由表。

router\_100\_101 的路由表内容如下：

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
10.10.1.0	*	255.255.255.0	U	0	0	0	qr-1
10.10.2.0	*	255.255.255.0	U	0	0	0	qr-2

router\_102\_103 的路由表内容如下：

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
10.10.1.0	*	255.255.255.0	U	0	0	0	qr-3
10.10.2.0	*	255.255.255.0	U	0	0	0	qr-4

这样的路由表是可以工作的。

例如，vlan102 上有数据包要发到 vlan103。选择路由时，会查看 router\_102\_103 的路由表，

匹配第二个条目，数据通过 qr-4 被正确地发送到 vlan103。

同样当 vlan100 上有数据包要发到 vlan101 时，会匹配 router\_100\_101 路由表的第二个条目，数据通过 qr-2 被正确地发送到 vlan101。

可见，namespace 使得每个 router 有自己的路由表，而且不会与其他 router 冲突，所以能很好地支持网络重叠。

### 3. 访问外网

通过 router 可以实现位于不同 vlan 中的 instance 之间的通信。

接下来要探讨的问题是 instance 如何与外部网络通信。这里的外部网络指的是租户网络以外的网络。租户网络是由 Neutron 创建和维护的网络。外部网络不由 Neutron 创建。如果是私有云，外部网络通常指的是公司 intranet；如果是公有云，外部网络通常指的是 internet。

具体到我们的实验网络环境：

计算节点和控制节点 eth1 提供的是租户网络，IP 段租户可以自由设置。

控制节点 eth2 连接的就是外部网络，IP 网段为 10.10.10.2/24，如图 9-147 所示。

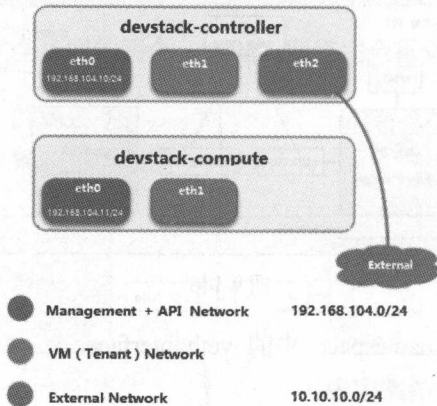


图 9-147

#### (1) 配置准备

为了连接外部网络，需要在配置文件中告诉 Neutron 外部网络的类型以及对应的物理网卡。因为外部网络是已经存在的物理网络，一般都是 flat 或者 vlan 类型。

这里我们将外部网络的 label 命名为“external”。如果类型为 flat，控制节点 /etc/neutron/plugins/ml2/ml2\_conf.ini，配置如图 9-148 所示。

```
[ml2_type_flat]
flat_networks = external
[linux_bridge]
physical_interface_mappings = default:eth1,external:eth2
```

图 9-148

如果类型为 vlan，配置如图 9-149 所示。

```
[ml2_type_vlan]
network_vlan_ranges = default:100:100,external
[linux_bridge]
physical_interface_mappings = default:eth1,external:eth2
```

图 9-149

修改配置后，需要重启 neutron 的相关服务。

在我们的网络环境中，外部网络是 flat 类型。

下面我们演示如何创建外部网络 ext\_net。

## (2) 创建 ext\_net

进入 Admin→Networks 菜单，单击“Create Network”按钮，如图 9-150 所示。

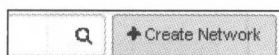


图 9-150

显示创建页面，如图 9-151 所示。

**Create Network**

Name:

Project \*:

Provider Network Type \*:

Physical Network \*:

Admin State \*:

☐ Shared

☒ External Network

**Description:**  
 Create a new network for any project as you need.  
 Provider specified network can be created. You can specify a physical network type (like Flat, VLAN, GRE, and VXLAN) and its segmentation\_id or physical network name for a new virtual network.  
 In addition, you can create an external network or a shared network by checking the corresponding checkbox.

图 9-151

Provider Network Type 选择“Flat”。

Physical Network 填写“external”，与 ml2\_conf.ini 中 flat\_networks 参数的设置保持一致。

选中 External Network 选择框。

单击“Create Network”，ext\_net 创建成功，如图 9-152 所示。

<input type="checkbox"/>	Project	Network Name	Subnets Associated
<input type="checkbox"/>	admin	ext_net	
<input type="checkbox"/>	admin	vlan101	subnet_172_16_101_0 172.16.101.0/24
<input type="checkbox"/>	admin	vlan100	subnet_172_16_100_0 172.16.100.0/24
Displaying 3 items			

图 9-152

单击“ext\_net”链接，进入 network 配置页面，目前还没有 subnet，单击“Create Subnet”按钮，如图 9-153 所示。

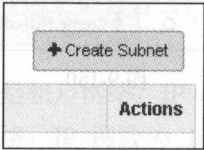


图 9-153

创建 subnet\_10\_10\_10\_0，IP 地址为 10.10.10.0/24，如图 9-154 所示。

### Create Subnet

Subnet

Subnet Details

Subnet Name

subnet\_10\_10\_10\_0

Network Address ⓘ

10.10.10.0/24

IP Version

IPv4

Gateway IP ⓘ

☐ Disable Gateway

« Back

Next »

Create a subnet associated with the network. Advanced configuration is available by clicking on the "Subnet Details" tab.

图 9-154

这里 Gateway 使用默认地址 10.10.10.1。通常我们需要询问网络管理员外网 subnet 的 Gateway IP，然后填在这里。

单击“Next”，如图 9-155 所示。



**Create Subnet**

Subnet Details

☒ Enable DHCP Specify additional attributes for the subnet.

Allocation Pools

DNS Name Servers

Host Routes

« Back Create

图 9-155

因为我们不会直接为 instance 分配外网 IP，所以一般不需要 enable DHCP。  
单击“Create”，如图 9-156 所示。

Name	CIDR	IP Version	Gateway IP
subnet_10_10_10_0	10.10.10.0/24	IPv4	10.10.10.1

Displaying 1 item

图 9-156

subnet 创建成功，网关为 10.10.10.1。

下面查看控制节点网络结构的变化，执行 `brctl show`，如图 9-157 所示。

```
root@devstack-controller:~# brctl show
bridge name      bridge id        STP enabled      interfaces
brq1d7040b8-01   8000.005056b0b5ff  no               eth1.101
                                                           tap5b1a2247-32
                                                           tape17162c5-00
brq3fcfdb98-9d   8000.005056b0b5ff  no               eth1.100
                                                           tap1180bbe8-06
                                                           tapd568ba1a-74
brqe496d3d2-53   8000.005056b0b52b  no               eth2
virbr0           8000.000000000000  yes
```

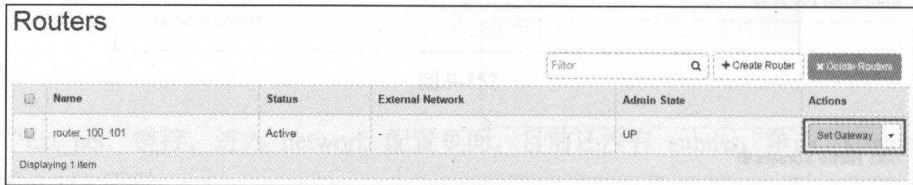
图 9-157

增加了一个网桥 brqe496d3d2-53，物理网卡 eth2 已经连接到该 bridge。

(3) 将 ext\_net 连接到 router\_100\_101

接下来需要将外网连接到 Neutron 的虚拟路由器，这样 instance 才能访问外网。

单击菜单 Project → Network → Routers 进入 router 列表，如图 9-158 所示。

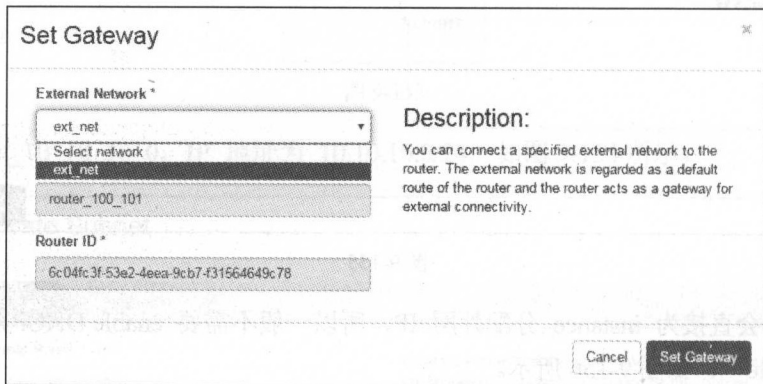


Name	Status	External Network	Admin State	Actions
router_100_101	Active		UP	Set Gateway

Displaying 1 item

图 9-158

单击 router\_100\_101 的 “Set Gateway” 按钮，如图 9-159 所示。



**Set Gateway**

External Network \*

ext\_net

Select network

ext\_net

router\_100\_101

Description:

You can connect a specified external network to the router. The external network is regarded as a default route of the router and the router acts as a gateway for external connectivity.

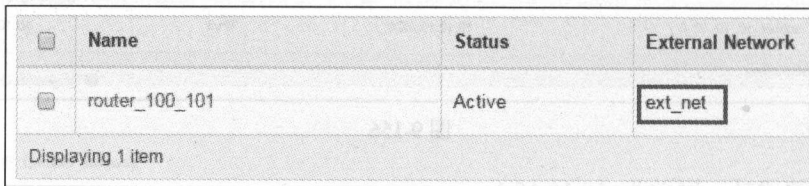
Router ID \*

6c04fc3f-53e2-4eea-9cb7-f31564649c78

Cancel Set Gateway

图 9-159

在 “External Network” 下拉列表中选择 ext\_net，单击 “Set Gateway”，如图 9-160 所示。



Name	Status	External Network
router_100_101	Active	ext_net

Displaying 1 item

图 9-160

外网设置成功。

我们需要看看 router 发生了什么变化。

单击 “router\_100\_101” 链接，打开 “Interfaces” 标签页，如图 9-161 所示。

Overview Interfaces Static Routes				
<input type="checkbox"/>	Name	Fixed IPs	Status	Type
<input type="checkbox"/>	(b8b32a88-03e3)	10.10.10.2	Active	External Gateway
<input type="checkbox"/>	(d568ba1a-740e)	172.16.100.1	Active	Internal Interface
<input type="checkbox"/>	(e17162c5-00fa)	172.16.101.1	Active	Internal Interface
Displaying 3 items				

图 9-161

router 多了一个新的 interface, IP 为 10.10.10.2。

该 interface 用于连接外网 ext\_net。

查看控制节点的网络结构, 外网 bridge 上已经连接了 router 的 tap 设备 tapb8b32a88-03, 如图 9-162 所示。

```

root@devstack-controller:~# brctl show
bridge name      bridge id        STP enabled     interfaces
brq1d7040b8-01   8000.005056b0b5ff  no              eth1.101
                                                          tap5b1a2247-32
                                                          tapc17162c5-00
brq3fcfdb98-9d   8000.005056b0b5ff  no              eth1.100
                                                          tap1180bbe8-06
                                                          tapd568ba1a-74
brqe496d3d2-53   8000.005056b0b52b  no              eth2
                                                          tapb8b32a88-03
virbr0           8000.000000000000  yes

```

图 9-162

在 router 的 namespace 中查看 tapb8b32a88-03 的 veth pair 设备, 如图 9-163 所示。

```

root@devstack-controller:~# ip netns exec grouter-6c04fc3f-53e2-4eea-9cb7-f31564649c78 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
   inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: gr-d568ba1a-74: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
   default qlen 1000
   link/ether fa:16:3e:07:92:a5 brd ff:ff:ff:ff:ff:ff
   inet 172.16.100.1/24 brd 172.16.100.255 scope global qr-d568ba1a-74
       valid_lft forever preferred_lft forever
   inet6 fe80::f816:3eff:fe07:92a5/64 scope link
       valid_lft forever preferred_lft forever
3: gr-e17162c5-00: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
   default qlen 1000
   link/ether fa:16:3e:89:15:ba brd ff:ff:ff:ff:ff:ff
   inet 172.16.101.1/24 brd 172.16.101.255 scope global qr-e17162c5-00
       valid_lft forever preferred_lft forever
   inet6 fe80::f816:3eff:fe89:15ba/64 scope link
       valid_lft forever preferred_lft forever
4: qg-b8b32a88-03: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
   default qlen 1000
   link/ether fa:16:3e:1a:c5:af brd ff:ff:ff:ff:ff:ff
   inet 10.10.10.2/24 brd 10.10.10.255 scope global qg-b8b32a88-03
       valid_lft forever preferred_lft forever
   inet6 fe80::f816:3eff:fela:c5af/64 scope link
       valid_lft forever preferred_lft forever
root@devstack-controller:~#

```

图 9-163

该 veth pair 命名为 qg-b8b32a88-03, 上面配置了 IP 10.10.10.2。

router 的每个 interface 在 namespace 中都有对应的 veth。

如果 veth 用于连接租户网络，命名格式为 qr-xxx，比如 qr-d568bala-74 和 qr-e17162c5-00。

如果 veth 用于连接外部网络，命名格式为 qg-xxx，比如 qg-b8b32a88-03。

查看 router 的路由表信息，如图 9-164 所示。

```
root@devstack-controller:~# ip netns exec qrouter-6c04fc3f-53e2-4eea-9cb7-f31564649c78 route
kernel IP routing table
Destination      Gateway         Genmask         Flags Metric Ref    Use Iface
default          10.10.10.1     0.0.0.0         UG    0     0        0 qg-b8b32a88-03
10.10.10.0       *              255.255.255.0   U    0     0        0 qg-b8b32a88-03
172.16.100.0     *              255.255.255.0   U    0     0        0 qr-d568bala-74
172.16.101.0     *              255.255.255.0   U    0     0        0 qr-e17162c5-00
```

图 9-164

可以看到默认网关为 10.10.10.1。

意味着对于访问 vlan100 和 vlan101 租户网络以外的所有流量，router\_100\_101 都将转发给 ext\_net 的网关 10.10.10.1。

现在 router\_100\_101 已经同时连接了 vlan100、vlan101 和 ext\_net 三个网络，如图 9-165 所示。

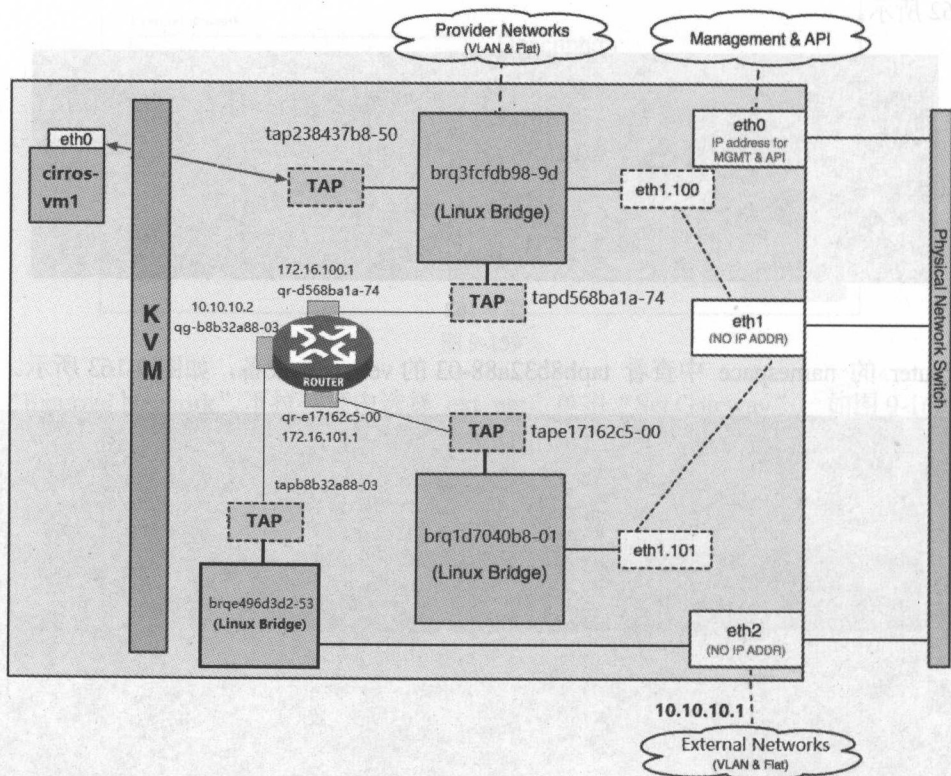


图 9-165

下面我们在 cirros-vm3 上测试一下，如图 9-166 所示。



```
$ ping 10.10.10.1
PING 10.10.10.1 (10.10.10.1): 56 data bytes
64 bytes from 10.10.10.1: seq=0 ttl=254 time=2.768 ms
64 bytes from 10.10.10.1: seq=1 ttl=254 time=1.705 ms
64 bytes from 10.10.10.1: seq=2 ttl=254 time=1.610 ms
64 bytes from 10.10.10.1: seq=3 ttl=254 time=1.495 ms
64 bytes from 10.10.10.1: seq=4 ttl=254 time=1.437 ms
64 bytes from 10.10.10.1: seq=5 ttl=254 time=2.272 ms
$
```

图 9-166

cirros-vm3 位于计算节点, 现在已经可以 Ping 到 ext\_net 网关 10.10.10.1 了。  
通过 traceroute 查看一下 cirros-vm3 到 10.10.10.1 的路径, 如图 9-167 所示。

```
$ traceroute 10.10.10.1
traceroute to 10.10.10.1 (10.10.10.1), 30 hops max, 46 byte packets
 1 host-172-16-101-1.openstacklocal (172.16.101.1) 0.763 ms 0.521 ms 0.760 ms
 2 10.10.10.1 (10.10.10.1) 2.678 ms 0.789 ms 2.157 ms
$
```

图 9-167

数据包经过两跳到达 10.10.10.1 网关。

- (1) 数据包首先发送到 router\_100\_101 连接 vlan101 的 interface (172.16.101.1)。
- (2) 然后通过连接 ext\_net 的 interface (10.10.10.2) 转发出去, 最后到达 10.10.10.1。

当数据包从 router 连接外网的接口 qg-b8b32a88-03 发出的时候, 会做一次 Source NAT, 即将包的源地址修改为 router 的接口地址 10.10.10.2, 这样就能够保证目的端能够将应答的包发回给 router, 然后再转发回源端 instance。

可以通过 iptables 命令查看 SNAT 的规则, 如图 9-168 所示。

```
root@devstack-controller:~# ip netns exec qrouter-6c04fc3f-53e2-4eea-9cb7-f31564649c78
iptables -t nat -S
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-N neutron-postrouting-bottom
-N neutron-vpn-agen-OUTPUT
-N neutron-vpn-agen-POSTROUTING
-N neutron-vpn-agen-PREROUTING
-N neutron-vpn-agen-float-snat
-N neutron-vpn-agen-snat
-A PREROUTING -j neutron-vpn-agen-PREROUTING
-A OUTPUT -j neutron-vpn-agen-OUTPUT
-A POSTROUTING -j neutron-vpn-agen-POSTROUTING
-A POSTROUTING -j neutron-postrouting-bottom
-A neutron-postrouting-bottom -m comment --comment "Perform source NAT on outgoing traffic." -j neutron-vpn-agen-snat
-A neutron-vpn-agen-POSTROUTING ! -i qg-b8b32a88-03 ! -o qg-b8b32a88-03 -m conntrack !
--ctstate DNAT -j ACCEPT
-A neutron-vpn-agen-PREROUTING -d 169.254.169.254/32 -i qr+ -p tcp -m tcp --dport 80 -
j REDIRECT --to-ports 9697
-A neutron-vpn-agen-snat -j neutron-vpn-agen-float-snat
-A neutron-vpn-agen-snat -o qg-b8b32a88-03 -j SNAT --to-source 10.10.10.2
-A neutron-vpn-agen-snat -m mark ! --mark 0x20xffff -m conntrack --ctstate DNAT -j SNA
T --to-source 10.10.10.2
```

图 9-168

当 cirros-vm3 (172.16.101.3) Ping 10.10.10.1 时, 可用通过 tcpdump 分别观察 router 两个 interface 的 icmp 数据包来验证 SNAT 的行为。

vlan101 interface qr-e17162c5-00 的 tcpdump 输出, 如图 9-169 所示。

```
root@devstack-controller:~# ip netns exec qrouter-6c04fc3f-53e2-4eea-9cb7-f31564649c78
tcpdump -i qr-e17162c5-00 -n icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on qr-e17162c5-00, link-type EN10MB (Ethernet), capture size 65535 bytes
17:49:24.82 IP [172.16.101.3] > 10.10.10.1: ICMP echo request, id 33537, seq 0, length 64
17:49:24.89 IP 10.10.10.1 > [172.16.101.3]: ICMP echo reply, id 33537, seq 0, length 64
```

图 9-169

ext\_net interface qg-b8b32a88-03 的 tcpdump 输出，如图 9-170 所示。

```
root@devstack-controller:~# ip netns exec qrouter-6c04fc3f-53e2-4eea-9cb7-f31564649c78
tcpdump -i qg-b8b32a88-03 -n icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on qg-b8b32a88-03, link-type EN10MB (Ethernet), capture size 65535 bytes
17:49:24.89 IP [10.10.10.2] > 10.10.10.1: ICMP echo request, id 33537, seq 0, length 64
17:49:24.89 IP 10.10.10.1 > [10.10.10.2]: ICMP echo reply, id 33537, seq 0, length 64
```

图 9-170

SNAT 让 instance 能够直接访问外网，但外网还不能直接访问 instance，因为 instance 没有外网 IP。

这里“直接访问 instance”是指通信连接由外网发起，例如从外网 SSH cirros-vm3。

这个问题可以通过 floating IP 解决。

#### 4. floating IP

当租户网络连接到 Neutron router，通常将 router 作为默认网关。

当 router 接收到 instance 的数据包，并将其转发到外网时：router 会修改包的源地址为自己的外网地址，这样确保数据包转发到外网，并能够从外网返回。

router 修改返回的数据包，并转发给真正的 instance。这个行为被称作 Source NAT。

如果需要从外网直接访问 instance，可以利用 floating IP。floating IP 提供静态 NAT 功能，建立外网 IP 与 instance 租户网络 IP 的一对一映射。floating IP 是配置在 router 提供网关的外网 interface 上的。

router 会根据通信的方向修改数据包的源或者目的地址。

下面我们通过实验深入学习 floating IP。

##### (1) 创建并分配 floating IP

单击 Project → Compute → Access & Security 菜单，打开 Floating IPs 标签页，如图 9-171 所示。

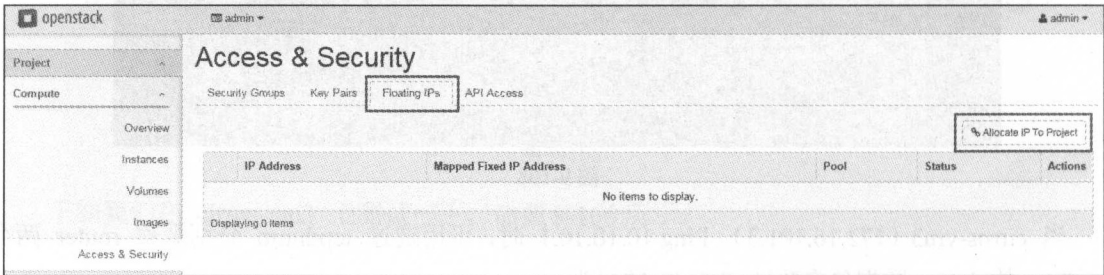


图 9-171

单击“Allocate IP To Project”按钮，如图 9-172 所示。

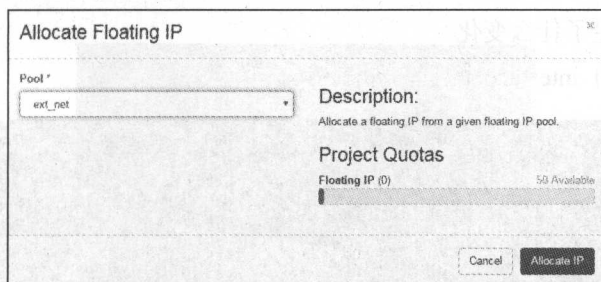


图 9-172

floating IP Pool 为 ext\_net，单击“Allocate IP”按钮，如图 9-173 所示。

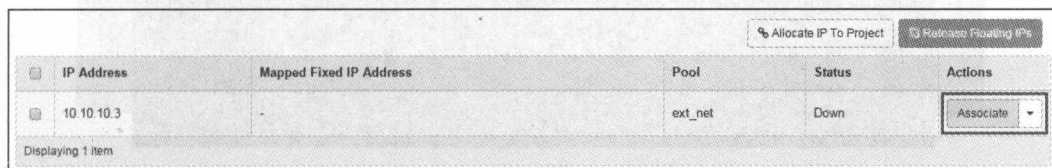


图 9-173

从 Pool 中成功分配了一个 IP 10.10.10.3。

下面我们将它分配给 cirros-vm3，单击“Associate”按钮，如图 9-174 所示。

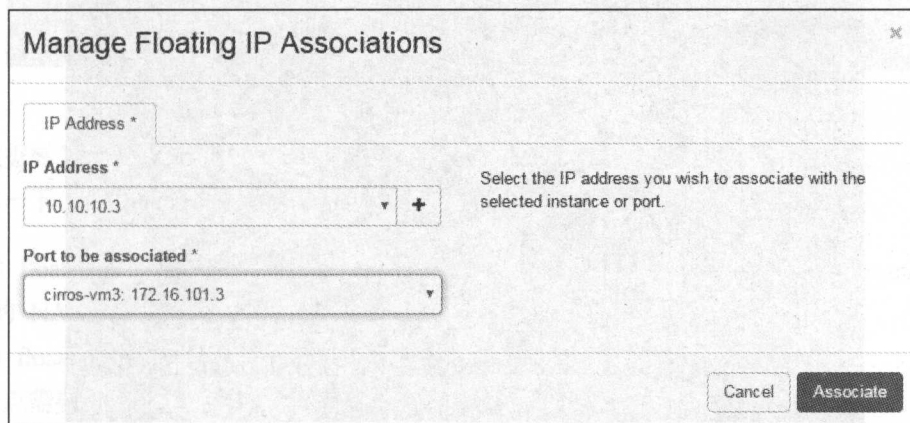


图 9-174

在下拉列表中选择 cirros-vm3，单击“Associate”按钮，如图 9-175 所示。

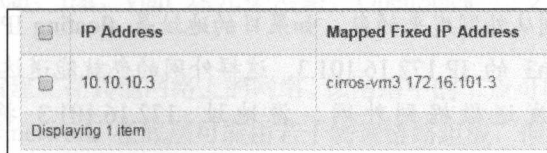


图 9-175

分配成功，floating IP 10.10.10.3 已经对应到 cirros-vm3 的租户 IP 172.16.101.3。

下面我们观察一下底层网络发生了怎样的变化。

## (2) 底层网络发生了什么变化

首先查看 router 的 interface 配置, 如图 9-176 所示。

```
root@devstack-controller:~# ip netns exec qrouter-6c04fc3f-53e2-4ee2-1b: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
valid_lft forever preferred_lft forever
inet6 ::1/128 scope host
valid_lft forever preferred_lft forever
2: qr-d568bala-74: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
link/ether fa:16:3e:07:92:a5 brd ff:ff:ff:ff:ff:ff
inet 172.16.100.1/24 brd 172.16.100.255 scope global qr-d568bala-74
valid_lft forever preferred_lft forever
inet6 fe80::f816:3eff:fe07:92a5/64 scope link
valid_lft forever preferred_lft forever
3: qr-e17162c5-00: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
link/ether fa:16:3e:89:15:ba brd ff:ff:ff:ff:ff:ff
inet 172.16.101.1/24 brd 172.16.101.255 scope global qr-e17162c5-00
valid_lft forever preferred_lft forever
inet6 fe80::f816:3eff:fe89:15ba/64 scope link
valid_lft forever preferred_lft forever
4: qq-b8b32a88-03: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
link/ether fa:16:3e:1a:c5:af brd ff:ff:ff:ff:ff:ff
inet 10.10.10.2/24 brd 10.10.10.255 scope global qq-b8b32a88-03
valid_lft forever preferred_lft forever
inet 10.10.10.3/32 brd 10.10.10.3 scope global qq-b8b32a88-03
valid_lft forever preferred_lft forever
inet6 fe80::f816:3eff:fe1a:c5af/64 scope link
valid_lft forever preferred_lft forever
```

图 9-176

可以看到, floating IP 已经配置到 router 的外网 interface qq-b8b32a88-03 上。

查看 router 的 NAT 规则, 如图 9-177 所示。

```
root@devstack-controller:~# ip netns exec qrouter-6c04fc3f-53e2-4eea-9cb7-f31564649c78: iptables -t nat -s
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-N neutron-postrouting-bottom
-N neutron-vpn-agen-OUTPUT
-N neutron-vpn-agen-POSTROUTING
-N neutron-vpn-agen-PREROUTING
-N neutron-vpn-agen-float-snat
-N neutron-vpn-agen-snat
-A PREROUTING -j neutron-vpn-agen-PREROUTING
-A OUTPUT -j neutron-vpn-agen-OUTPUT
-A POSTROUTING -j neutron-vpn-agen-POSTROUTING
-A POSTROUTING -j neutron-postrouting-bottom
-A neutron-postrouting-bottom -m comment --comment "Perform source NAT on outgoing traf
fic." -j neutron-vpn-agen-snat
-A neutron-vpn-agen-OUTPUT -d 10.10.10.3/32 -j DNAT --to-destination 172.16.101.3
-A neutron-vpn-agen-POSTROUTING -j qq-b8b32a88-03 -o qq-b8b32a88-03 -m conntrack --ctstate DNAT -j ACCEPT
-A neutron-vpn-agen-PREROUTING -d 169.254.169.254/32 -i qr++ -p tcp -m tcp --dport 80 -j REDIRECT --to-ports 9697
-A neutron-vpn-agen-PREROUTING -d 10.10.10.3/32 -j DNAT --to-destination 172.16.101.3
-A neutron-vpn-agen-float-snat -s 172.16.101.3/32 -j SNAT --to-source 10.10.10.3
-A neutron-vpn-agen-snat -o qq-b8b32a88-03 -j SNAT --to-source 10.10.10.2
-A neutron-vpn-agen-snat -m mark ! --mark 0x2/0xffff -m conntrack --ctstate DNAT -j SNAT --to-source 10.10.10.2
```

图 9-177

iptables 增加了两条处理 floating IP 的规则:

- 当 router 接收到从外网发来的包, 如果目的地址是 floating IP 10.10.10.3, 将目的地址修改为 cirros-vm3 的 IP 172.16.101.3。这样外网的包就能送达到 cirros-vm3。
- 当 cirros-vm3 发送数据到外网, 源地址 172.16.101.3 将被修改为 floating IP 10.10.10.3。

下面我们通过 PING 测试一下。



在我的实验环境中, 10.10.10.1 是外网中的物理交换机, 现在让它 PING cirros-vm3, 如图 9-178 所示。

```
Switch-01 (config)#ping 10.10.10.3 data-port
Connecting via DATA port.
[host 10.10.10.3, max tries 5, delay 1000 msec]
10.10.10.3: #1 ok, RTT 2 msec.
10.10.10.3: #2 ok, RTT 2 msec.
10.10.10.3: #3 ok, RTT 4 msec.
10.10.10.3: #4 ok, RTT 1 msec.
10.10.10.3: #5 ok, RTT 3 msec.
```

图 9-178

能够 PING 通。

我们通过 tcpdump 可用在 router 的 interface 上观察 floating IP 的行为。

ext\_net interface qg-b8b32a88-03 的 tcpdump 输出, 如图 9-179 所示。

```
root@devstack-controller:~# ip netns exec qrouter-6c04fc3f-53e2-4eea-9cb7-f31564649c78
tcpdump -i qg-b8b32a88-03 -n icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on qg-b8b32a88-03, link-type EN10MB (Ethernet), capture size 65535 bytes
18:53:39.47 IP 10.10.10.1 > 10.10.10.3: ICMP echo request, id 60, seq 1, length 8
18:53:39.48 IP 10.10.10.3 > 10.10.10.1: ICMP echo reply, id 60, seq 1, length 8
```

图 9-179

可见, 在外网接口 qg-b8b32a88-03 上, 始终是通过 floating IP 10.10.10.3 与外网通信。

vlan101 interface qr-e17162c5-00 的 tcpdump 输出, 如图 9-180 所示。

```
root@devstack-controller:~# ip netns exec qrouter-6c04fc3f-53e2-4eea-9cb7-f31564649c78
tcpdump -i qr-e17162c5-00 -n icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on qr-e17162c5-00, link-type EN10MB (Ethernet), capture size 65535 bytes
18:53:39.47 IP 10.10.10.1 > 172.16.101.3: ICMP echo request, id 60, seq 1, length 8
18:53:39.48 IP 172.16.101.3 > 10.10.10.1: ICMP echo reply, id 60, seq 1, length 8
```

图 9-180

当数据转发到租户网络, 地址已经变为 cirros-vm3 的租户 IP 172.16.101.3 了。

小结一下:

(1) floating IP 能够让外网直接访问租户网络中的 instance。这是通过在 router 上应用 iptables 的 NAT 规则实现的。

(2) floating IP 是配置在 router 的外网 interface 上的, 而非 instance。

这一点需要特别注意。

## 9.4.9 vxlan network

除了前面讨论的 local、flat、vlan 这几类网络, OpenStack 还支持 vxlan 和 gre 这两种 overlay network。

overlay network 是指建立在其他网络上的网络。该网络中的节点可以看作通过虚拟(或逻辑)链路连接起来的。overlay network 在底层可能由若干物理链路组成, 但对于节点, 不需要关心这些底层实现。例如 P2P 网络就是 overlay network, 隧道也是。vxlan 和 gre 都是基于隧道技术实现的, 它们也都是 overlay network。

目前 Linux Bridge 只支持 vxlan，不支持 gre。Open Vswitch 两者都支持。  
vxlan 与 gre 实现非常类似，而且 vxlan 用得较多，所以本教程只介绍 vxlan。

1. VXLAN 概念

VXLAN 全称 Virtual eXtensible Local Area Network。正如名字所描述的，VXLAN 提供与 VLAN 相同的以太网二层服务，但是拥有更强的扩展性和灵活性。与 VLAN 相比，VXLAN 有下面几个优势：

- 支持更多的二层网段。

VLAN 使用 12-bit 标记 VLAN ID，最多支持 4094 个 VLAN，这对于大型云部署会成为瓶颈。VXLAN 的 ID（VNI 或者 VNID）则用 24-bit 标记，支持 16777216 个二层网段。

- 能更好地利用已有的网络路径。

VLAN 使用 Spanning Tree Protocol 避免环路，这会导致有一半的网络路径被 block 掉。VXLAN 的数据包是封装到 UDP 通过三层传输和转发的，可以使用所有的路径。

- 避免物理交换机 MAC 表耗尽。

由于采用隧道机制，TOR (Top on Rack) 交换机无须在 MAC 表中记录虚拟机的信息。

(1) VXLAN 封装和包格式

VXLAN 是将二层建立在三层上的网络。通过将二层数据封装到 UDP 的方式来扩展数据中心的二层网段数量。VXLAN 是一种在现有物理网络设施中支持大规模多租户网络环境的解决方案。VXLAN 的传输协议是 IP + UDP。

VXLAN 定义了一个 MAC-in-UDP 的封装格式。在原始的 Layer 2 网络包前加上 VXLAN header，然后放到 UDP 和 IP 包中。通过 MAC-in-UDP 封装，VXLAN 能够在 Layer 3 网络上建立起了一条 Layer 2 的隧道。

VXLAN 包的格式，如图 9-181 所示。

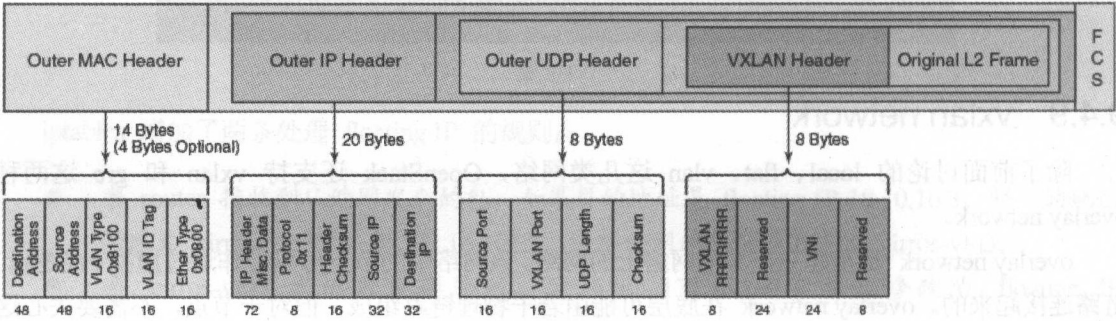


图 9-181

如图 9-181 所示，VXLAN 引入了 8-byte VXLAN header，其中 VNI 占 24-bit。VXLAN 和

原始的 L2 frame 被封装到 UDP 包中。这 24-bit 的 VNI 用于标示不同的二层网段，能够支持 16777216 个 LAN。

### (2) VXLAN Tunnel Endpoint

VXLAN 使用 VXLAN tunnel endpoint (VTEP) 设备处理 VXLAN 的封装和解封。

每个 VTEP 有一个 IP interface，配置了一个 IP 地址。VTEP 使用该 IP 封装 Layer 2 frame，并通过该 IP interface 传输和接收封装后的 VXLAN 数据包。

图 9-182 所示是 VTEP 的示意图。

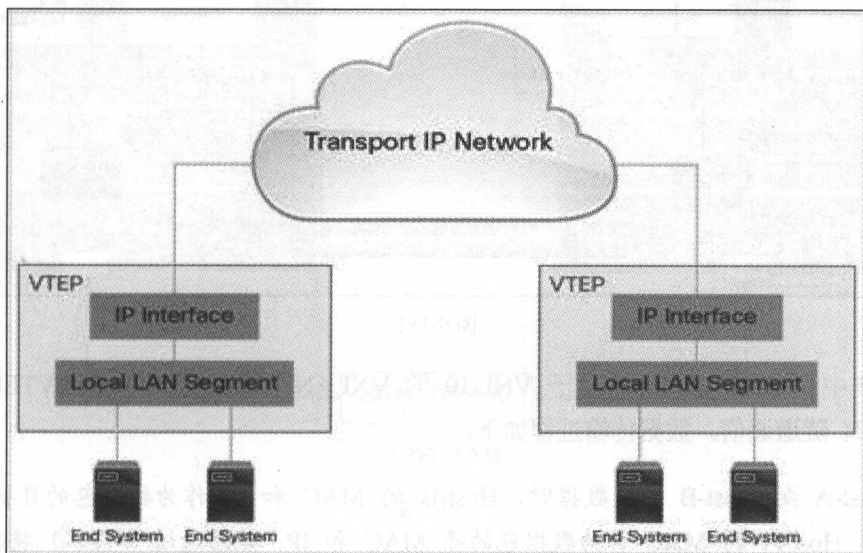


图 9-182

VXLAN 独立于底层的网络拓扑，反过来，两个 VTEP 之间的底层 IP 网络也独立于 VXLAN。

VXLAN 数据包是根据外层的 IP header 路由的，该 header 将两端的 VTEP IP 作为源和目标 IP。

### (3) VXLAN 包转发流

VXLAN 在 VTEP 间建立隧道，通过 Layer 3 网络传输封装后的 Layer 2 数据。

下面通过一个例子说明数据如何在 VXLAN 上传输，如图 9-183 所示。

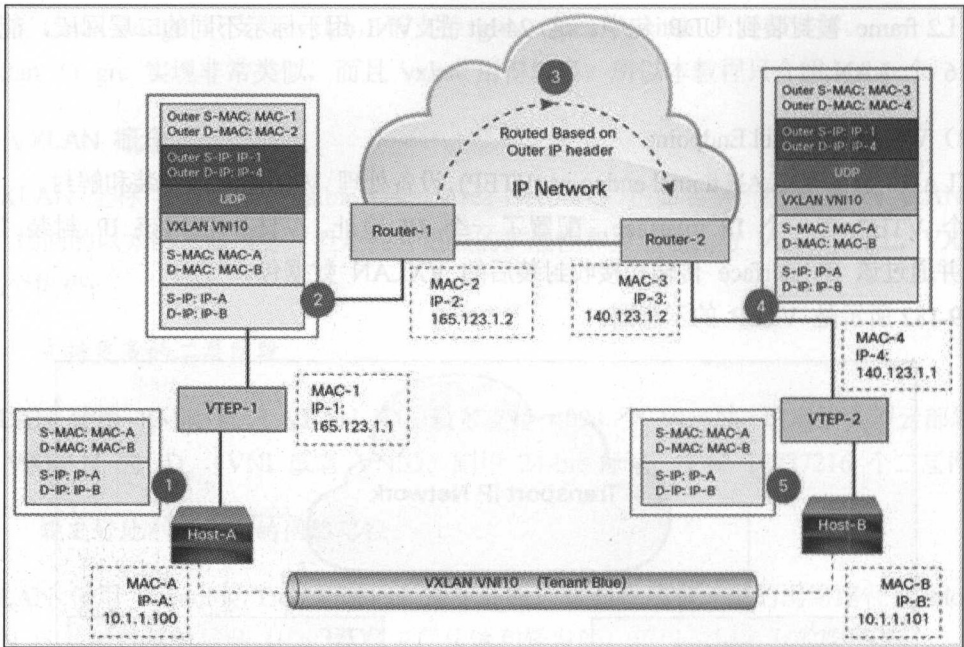


图 9-183

图 9-183 中 Host-A 和 Host-B 位于 VNI 10 的 VXLAN，通过 VTEP-1 和 VTEP-2 之间建立的 VXLAN 隧道通信。数据传输过程如下：

- Host-A 向 Host-B 发送数据时，Host-B 的 MAC 和 IP 作为数据包的目标 MAC 和 IP，Host-A 的 MAC 作为数据包的源 MAC 和 IP，然后通过 VTEP-1 将数据发送出去。
- VTEP-1 从自己维护的映射表中找到 MAC-B 对应的 VTEP-2，然后执行 VXLAN 封装，加上 VXLAN 头，UDP 头，以及外层 IP 和 MAC 头。此时的外层 IP 头，目标地址为 VTEP-2 的 IP，源地址为 VTEP-1 的 IP。同时由于下一跳是 Router-1，所以外层 MAC 头中目标地址为 Router-1 的 MAC。
- 数据包从 VTEP-1 发送出去后，外部网络的路由器会依据外层 IP 头进行包路由，最后到达与 VTEP-2 连接的路由器 Router-2。
- Router-2 将数据包发送给 VTEP-2。VTEP-2 负责解封数据包，依次去掉外层 MAC 头，外层 IP 头，UPD 头和 VXLAN 头。
- VTEP-2 依据目标 MAC 地址将数据包发送给 Host-B。

上面的流程我们看到 VTEP 是 VXLAN 的最核心组件，负责数据的封装和解封。隧道也是建立在 VTEP 之间的，VTEP 负责数据的传送。

#### (4) Linux 对 VXLAN 的支持

VTEP 可以由专有硬件来实现，也可以使用纯软件实现。

目前比较成熟的 VTEP 软件实现包括：



- 带 VXLAN 内核模块的 Linux
- Open vSwitch

我们先来看 Linux 如何支持 VXLAN，如图 9-184 所示。Open vSwitch 方式将在后面章节讨论。

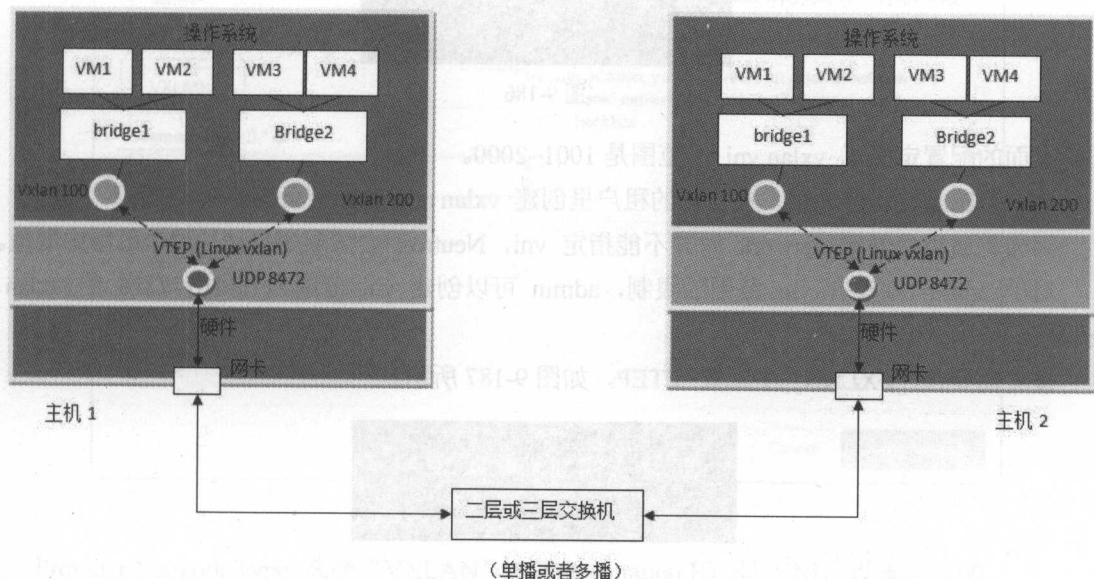


图 9-184

实现方式：

- Linux vxlan 创建一个 UDP Socket，默认在 8472 端口监听。
- Linux vxlan 在 UDP socket 上接收到 vxlan 包后，解包，然后根据其中的 vxlan ID 将它转给某个 vxlan interface，然后再通过它所连接的 linux bridge 转给虚拟机。
- Linux vxlan 在收到虚拟机发来的数据包后，将其封装为多播 UDP 包，从网卡发出。

到这里，相信大家对 VXLAN 的原理已经有了大致的了解。接下来我们将学习如何在 Neutron 中配置和实施 VXLAN。

## 2. 在 ML2 配置中 enable vxlan network

在 /etc/neutron/plugins/ml2/ml2\_conf.ini 设置 vxlan network 相关参数，如图 9-185 所示。

```
[ml2]
tenant_network_types = vxlan
extension_drivers = port_security
type_drivers = local,flat,vlan,qre,vxlan
mechanism_drivers = linuxbridge,l2population
```

图 9-185

```
tenant_network_types = vxlan
```

指定普通用户创建的网络类型为 vxlan。

这里还使用了一个名为 “l2population” 的 mechanism driver，我们放到后面单独介绍。

然后指定 vxlan 的范围，如图 9-186 所示。

```
[ml2_type_vxlan]
vni_ranges = 1001:2000
```

图 9-186

上面的配置定义了 vxlan vni 的范围是 1001~2000。

这个范围是针对普通用户在自己的租户里创建 vxlan network 的范围。

因为普通用户创建 network 时并不能指定 vni，Neutron 会按顺序自动从这个范围中取值。

对于 admin 则没有 vni 范围的限制，admin 可以创建 vni 范围为 1~16777216 的 vxlan network。

接着需要在 [VXLAN] 中配置 VTEP，如图 9-187 所示。

```
[VXLAN]
enable_vxlan = True
l2_population = True
local_ip = 166.66.16.10
```

图 9-187

local\_ip 指定 VTEP 的 IP 地址。

devstack\_controller 的 VTEP IP 是 166.66.16.10，网卡为 eth1。

devstack\_compute01 的 VTEP IP 是 166.66.16.11，网卡为 eth1，如图 9-188 所示。

```
[VXLAN]
enable_vxlan = True
l2_population = True
local_ip = 166.66.16.11
```

图 9-188

### 3. 创建 vxlan network “vxlan100\_net”

打开菜单 Admin → Networks，单击 “Create Network” 按钮，如图 9-189 所示。

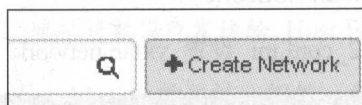
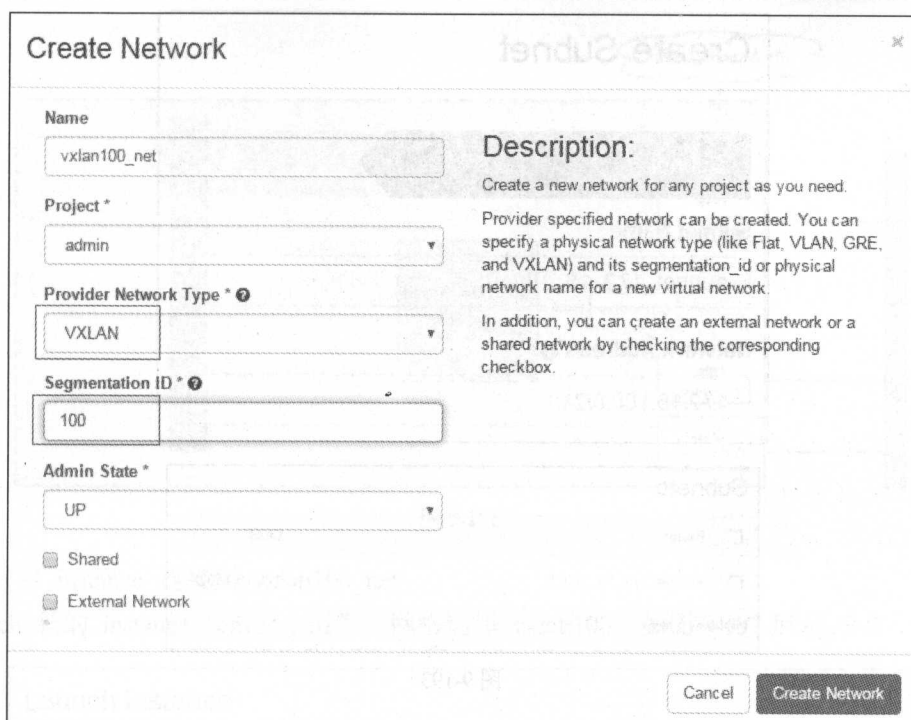


图 9-189

显示创建页面，如图 9-190 所示。



**Create Network**

**Name**  
vxlan100\_net

**Project \***  
admin

**Provider Network Type \***  
VXLAN

**Segmentation ID \***  
100

**Admin State \***  
UP

☐ Shared  
☐ External Network

**Description:**  
Create a new network for any project as you need.  
Provider specified network can be created. You can specify a physical network type (like Flat, VLAN, GRE, and VXLAN) and its segmentation\_id or physical network name for a new virtual network.  
In addition, you can create an external network or a shared network by checking the corresponding checkbox.

Cancel Create Network

图 9-190

Provider Network Type 选择“VXLAN”，Segmentation ID 即 VNI，设置为 100。  
单击“Create Network”，vxlan100 创建成功，如图 9-191 所示。

<input type="checkbox"/>	Project	Network Name	Subnets Associated
<input type="checkbox"/>	admin	vxlan100_net	
Displaying 1 item			

图 9-191

单击 vxlan100 链接，进入 network 配置页面，目前还没有 subnet，单击“Create Subnet”按钮，如图 9-192 所示。

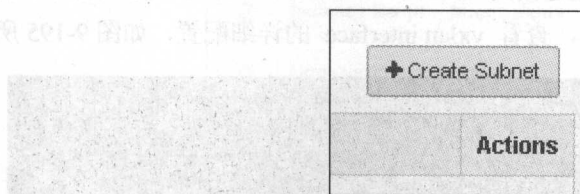


图 9-192

创建 subnet\_172\_16\_100\_0，IP 地址为 172.16.100.0/24，如图 9-193 所示。

### Create Subnet

Subnet

Subnet Details

Subnet Name

subnet\_172\_16\_100\_0

Network Address

172.16.100.0/24

Subnets

<input type="checkbox"/>	Name	CIDR
<input type="checkbox"/>	subnet_172_16_100_0	172.16.100.0/24

Displaying 1 item

图 9-193

(1) 底层网络发生了什么变化

在控制节点上执行 `brctl show`，查看当前的网络结构，如图 9-194 所示。

```
root@devstack-controller:~# brctl show
bridge name      bridge id        STP enabled      interfaces
brq1762d312-d4   8000.125b1fe3a9e5 no                tap4df76d0e-59
virbr0           8000.000000000000 yes               vxlan-100
root@devstack-controller:~#
```

图 9-194

Neutron 创建了：

- vxlan100 对应的网桥 `brq1762d312-d4`。
- vxlan interface `vxlan-100`。
- dhcp 的 tap 设备 `tap4df76d0e-59`。

`vxlan-100` 和 `tap4df76d0e-59` 已经连接到 `brq1762d312-d4`，`vxlan100` 的二层网络就绪。执行 `ip -d link show dev vxlan-100`，查看 `vxlan interface` 的详细配置，如图 9-195 所示。

```
root@devstack-controller:~# ip -d link show dev vxlan-100
8: vxlan-100: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue
    master brq1762d312-d4 state UNKNOWN mode DEFAULT group default
    link/ether a2:2b:d1:86:9b:12 brd ff:ff:ff:ff:ff:ff promiscuity 1
    vxlan id 100 dev eth1 port 0 0 proxy ageing 300
root@devstack-controller:~#
```

图 9-195

可见，`vxlan-100` 的 VNI 是 100，对应的 VTEP 网络接口为 `eth1`。此时 `vxlan100` 结构如图 9-196 所示。



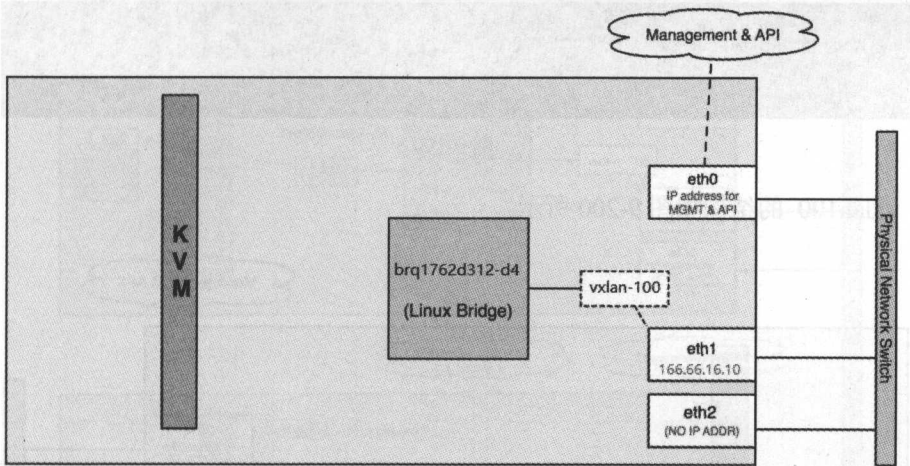


图 9-196

(2) 将 instance 连接到 vxlan100\_net

launch 新的 instance “cirros-vm1”，网络选择 vxlan100，如图 9-197 所示。

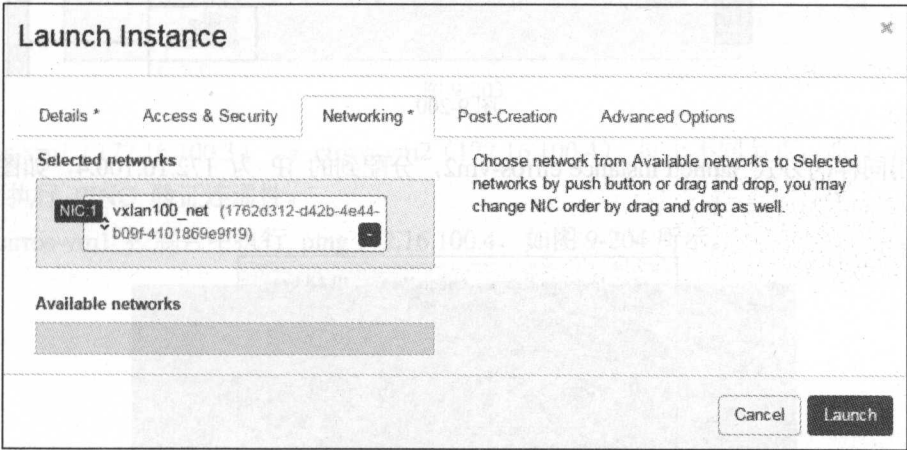


图 9-197

cirros-vm1 分配到的 IP 为 172.16.100.3，如图 9-198 所示。

<input type="checkbox"/>	Instance Name	Image Name	IP Address
<input type="checkbox"/>	cirros-vm1	cirros	172.16.100.3
Displaying 1 item			

图 9-198

cirros-vm1 被 schedule 到控制节点，对应的 tap 设备为 tap099caa87-cd，并且连接到 bridge brq1762d312-d4，如图 9-199 所示。

```
root@devstack-controller:~# brctl show
bridge name      bridge id        STP enabled  interfaces
brq1762d312-d4   8000.7a5c7aa665ec  no          tap099caa87-cd
                                                         tap4df76d0e-59
                                                         vxlan-100
virbr0           8000.000000000000  yes
```

图 9-199

当前 vxlan100 的结构如图 9-200 所示。

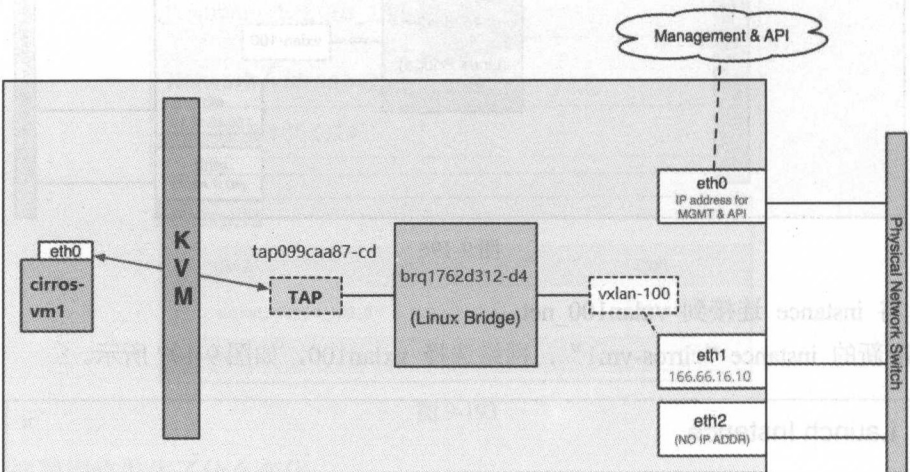


图 9-200

继续用同样的方式 launch instance cirros-vm2，分配到的 IP 为 172.16.100.4，如图 9-201 所示。

	Instance Name	Image Name	IP Address
<input checked="" type="checkbox"/>	cirros-vm2	cirros	172.16.100.4
<input checked="" type="checkbox"/>	cirros-vm1	cirros	172.16.100.3
Displaying 2 items			

图 9-201

cirros-vm2 被 schedule 到计算节点，对应的 tap 设备为 tap457cc048-aa，并且连接到 bridge brq1762d312-d4，如图 9-202 所示。

```
root@devstack-compute1:~# brctl show
bridge name      bridge id        STP enabled  interfaces
brq1762d312-d4   8000.3ec9d47188ad  no          tap457cc048-aa
                                                         vxlan-100
virbr0           8000.000000000000  yes
```

图 9-202

因为计算节点上没有 hdcp 服务，所以没有相应的 tap 设备。  
另外，bridge 的名称与控制节点上一致，都是 brq1762d312-d4，表明是同一个 network。  
当前 vxlan100 的结构如图 9-203 所示。

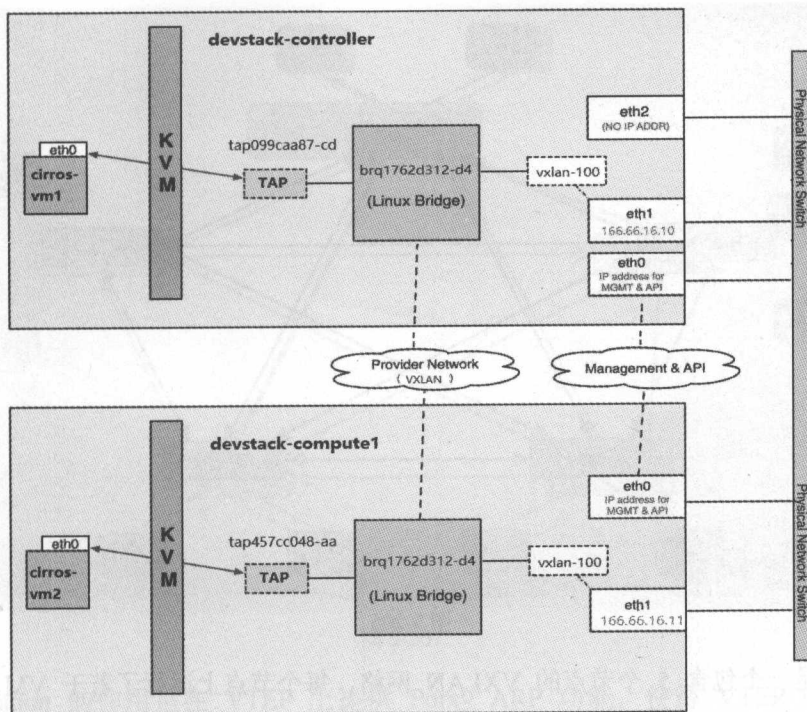


图 9-203

cirros-vm1 (172.16.100.3) 与 cirros-vm2 (172.16.100.4) 位于不同节点, 通过 vxlan100 相连, 下面执行 PING 验证连通性。

在 cirros-vm1 控制台中执行 ping 172.16.100.4, 如图 9-204 所示。

```
$ ping 172.16.100.4
PING 172.16.100.4 (172.16.100.4): 56 data bytes
64 bytes from 172.16.100.4: seq=0 ttl=64 time=2.603 ms
64 bytes from 172.16.100.4: seq=1 ttl=64 time=1.730 ms
64 bytes from 172.16.100.4: seq=2 ttl=64 time=1.804 ms
64 bytes from 172.16.100.4: seq=3 ttl=64 time=1.751 ms
64 bytes from 172.16.100.4: seq=4 ttl=64 time=1.912 ms
64 bytes from 172.16.100.4: seq=5 ttl=64 time=1.977 ms
64 bytes from 172.16.100.4: seq=6 ttl=64 time=1.537 ms
```

图 9-204

如我们预料, ping 成功。

对于多 vxlan 之间的 routing 以及 floating ip, 实现方式与 vlan 非常类似, 这里不再赘述, 请参看前面 vlan 相关章节。

#### 4. 理解 L2 Population

##### (1) L2 Population 原理

L2 Population 是用来提高 VXLAN 网络的 Scalability 的。

通常我们说某个系统的 Scalability 好, 其意思是: 当系统的规模变大时, 仍然能够高效地工作。回到 VXLAN 的场景, L2 Population 到底解决了怎样的 Scalability 问题? 请看图 9-205 所示。

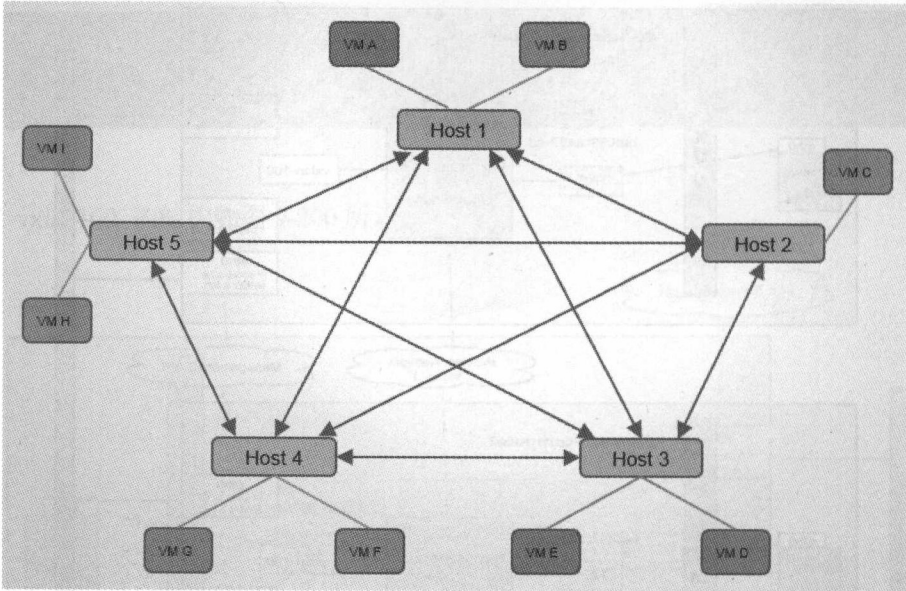


图 9-205

图 9-205 是一个包含 5 个节点的 VXLAN 网络，每个节点上运行了若干 VM。

现在假设 Host 1 上的 VM A 想与 Host 4 上的 VM G 通信，VM A 要做的第一步是获知 VM G 的 MAC 地址。

于是 VM A 需要在整个 VXLAN 网络中广播 APR 报文：“VM G 的 MAC 地址是多少？”，如图 9-206 所示。

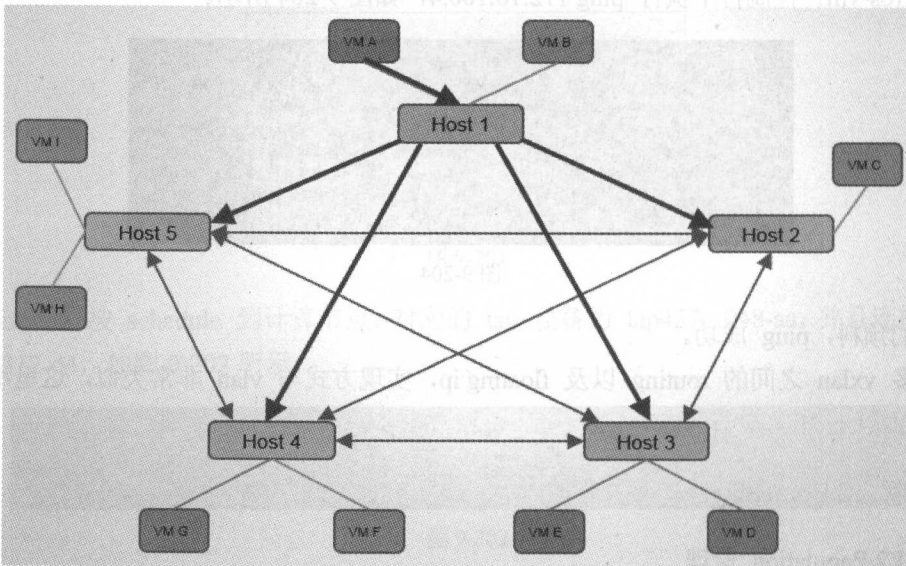


图 9-206

如果 VXLAN 网络的节点很多，上面广播的成本会很大，这样 Scalability 就成问题了。这时 L2 Population 出现了，如图 9-207 所示。



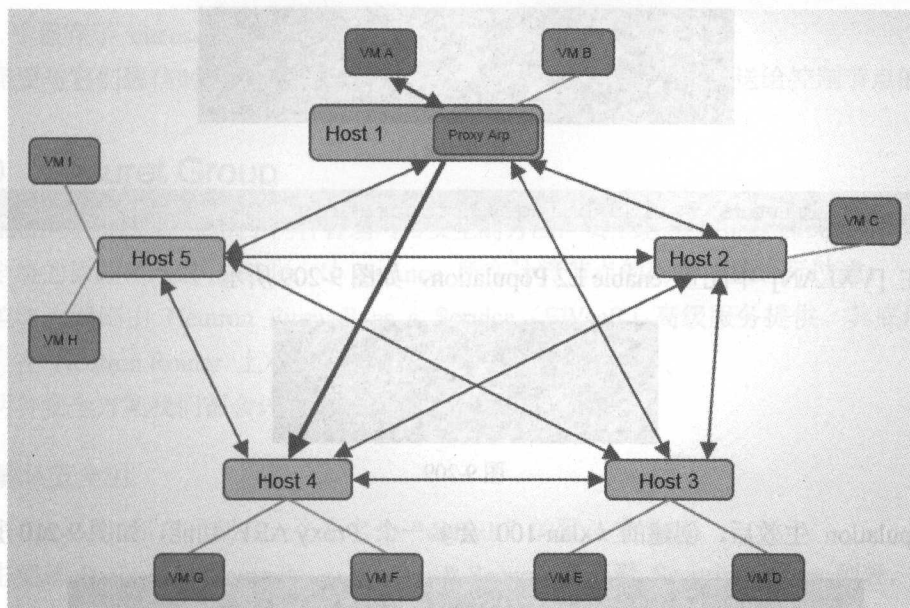


图 9-207

L2 Population 的作用是在 VTEP 上提供 Proxy ARP 功能, 使得 VTEP 能够预先获知 VXLAN 网络中的下面的信息:

- VM IP - MAC 对应关系
- VM - VTEP 的对应关系

当 VMA 需要与 VM G 通信时:

- Host 1 上的 VTEP 直接响应 VMA 的 ARP 请求, 告之 VM G 的 MAC 地址。
- 因为 Host 1 上的 VTEP 知道 VM G 位于 Host 4, 会将封装好的 VXLAN 数据包直接发送给 Host 4 的 VTEP。

这样就解决了 MAC 地址学习和 ARP 广播的问题, 从而保证了 VXLAN 的 Scalability。

那么下一个关键问题是:

VTEP 是如何提前获知 IP - MAC - VTEP 相关信息的呢? 其原理是:

Neutron 中保存每一个 port 的状态, 而 port 保存了 IP, MAC 相关数据。

instance 启动时, 其 port 状态会 from down 到 build 到 active。

因此, 每次 port 发生状态变化时, Neutron 都会通过 RPC 消息通知各节点上的 Neutron agent, 使得 VTEP 能够更新相关信息, 从而避免了不必要的隧道连接和广播。

## (2) L2 Population 配置

目前 L2 Population 支持 VXLAN with Linux bridge 和 VXLAN/GRE with OVS。

可以通过以下配置启用 L2 Population。

在 /etc/neutron/plugins/ml2/ml2\_conf.ini 设置 l2population mechanism driver, 如图 9-208 所示。

```

[devstack]
enable_network_types = [vxlan]
extension_drivers = port_security
type_drivers = local,flat,vlan,gre,vxlan
mechanism_drivers = [linuxbridge,l2population]

```

图 9-208

```
mechanism_drivers = linuxbridge,l2population
```

同时在 [VXLAN] 中配置 enable L2 Population, 如图 9-209 所示。

```

[libvirt]
enable_vxlan = True
l2_population = True
local_ip =

```

图 9-209

L2 Population 生效后, 创建的 vxlan-100 会多一个 Proxy ARP 功能, 如图 9-210 所示。

```

root@devstack-controller:~# ip -d link show dev vxlan-100
8: vxlan-100: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue
    link/ether a2:2b:d1:86:9b:12 brd ff:ff:ff:ff:ff:ff promiscuity 1
    vxlan id 100 dev eth1 port 0 0 proxy ageing 300
root@devstack-controller:~#

```

图 9-210

查看控制节点上的 forwarding database, 可以看到 VTEP 保存了 cirros-vm2 的 port 信息, 如图 9-211 所示。

```

root@devstack-controller:~# bridge fdb show dev vxlan-100
a2:2b:d1:86:9b:12 vlan 1 permanent
a2:2b:d1:86:9b:12 permanent
33:33:00:00:00:01 self permanent
01:00:5e:00:00:01 self permanent
33:33:ff:86:9b:12 self permanent
00:00:00:00:00:00 dst 166.66.16.11 self permanent
fa:16:3e:1d:23:a3 dst 166.66.16.11 self permanent
root@devstack-controller:~#

```

图 9-211

cirros-vm2 的 MAC 为 fa:16:3e:1d:23:a3。

VTEP IP 为 166.66.16.11。

当需要与 cirros-vm2 通信时, 控制节点 VTEP 166.66.16.10 会将封装好的 VXLAN 数据包直接发送给计算节点的 VTEP 166.66.16.11。

我们再查看一下计算节点上的 forwarding database, 如图 9-212 所示。

```

root@devstack-compute1:~# bridge fdb show dev vxlan-100
3e:c9:d4:71:88:ad vlan 1 permanent
3e:c9:d4:71:88:ad permanent
33:33:00:00:00:01 self permanent
01:00:5e:00:00:01 self permanent
33:33:ff:71:88:ad self permanent
00:00:00:00:00:00 dst 166.66.16.10 self permanent
fa:16:3e:22:bc:d8 dst 166.66.16.10 self permanent
fa:16:3e:16:e2:3e dst 166.66.16.10 self permanent
root@devstack-compute1:~#

```

图 9-212

fdb 中保存了 cirros-vm1 和 dhcp 的 port 信息。

当需要与它们通信时，计算节点 VTEP 知道应该将数据包直接发送给控制节点的 VTEP。

### 9.4.10 Securet Group

Neutron 为 instance 提供了两种管理网络安全的方法：安全组（Security Group）和虚拟防火墙。安全组的原理是通过 iptables 对 instance 所在计算节点的网络流量进行过滤。

虚拟防火墙则由 Neutron Firewall as a Service (FWaaS) 高级服务提供。其底层也是使用 iptables，在 Neutron Router 上对网络包进行过滤。

这两种安全方案我们都会讨论，本章先重点学习安全组。

#### 1. 默认安全组

每个 Project（租户）都有一个命名为“default”的默认安全组。

单击菜单 Project → Compute → Access & Security，查看 Security Group 列表，如图 9-213 所示。

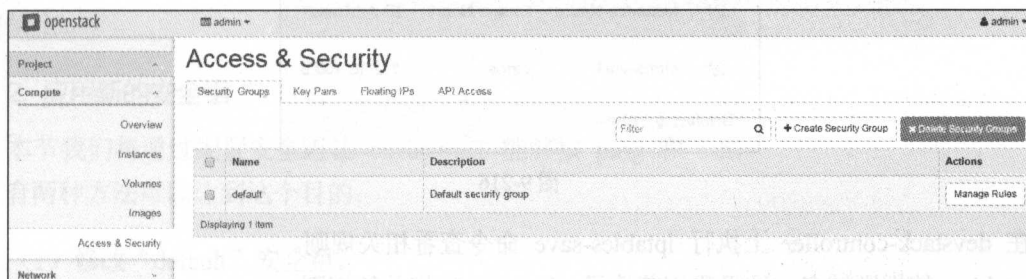


图 9-213

单击“Manage Rules”，查看“default”安全组的规则，如图 9-214 所示。

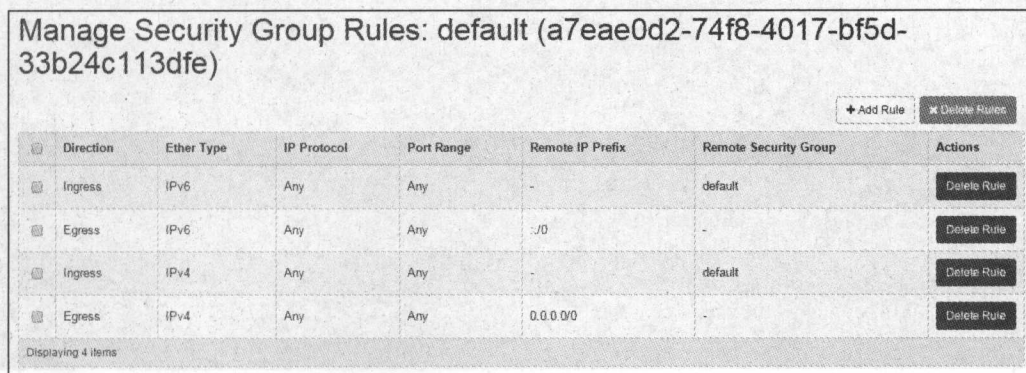


图 9-214

“default”安全组有四条规则，其作用是：

允许所有外出（Egress）的流量，但禁止所有进入（Ingress）的流量。

当我们创建 instance 时，可以在“Access & Security”标签页中选择安全组。

如果当前只有“default”这一个安全组，则会强制使用“default”，如图 9-215 所示。

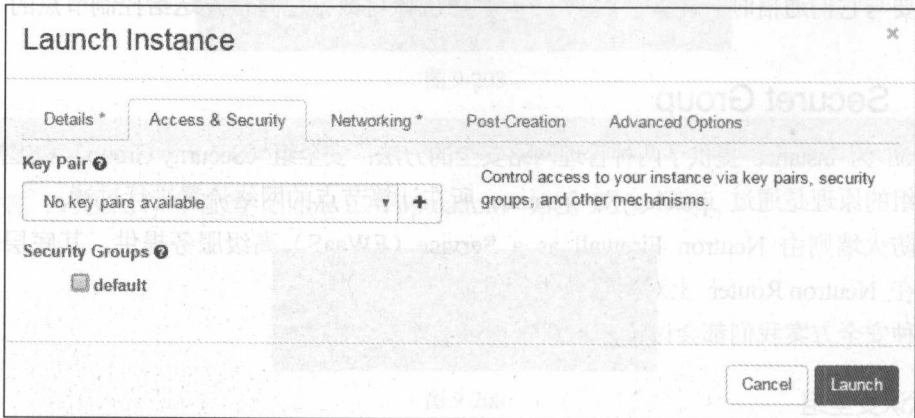


图 9-215

当前在 devstack-controller 上有 instance “cirros-vm1”，如图 9-216 所示。

<input type="checkbox"/>	Instance Name	Image Name	IP Address
<input type="checkbox"/>	cirros-vm1	cirros	172.16.100.3
Displaying 1 item			

图 9-216

在 devstack-controller 上执行 iptables-save 命令查看相关规则。

iptables 的规则较多，这里我们节选了 cirros-vm1 相关的规则。

如果大家想深入理解 iptables，可 google 相关文档，如图 9-217 所示。

```
-A neutron-linuxbri-18bca5b86-2 -s 172.16.100.2/32 -p udp --sport 67 -m udp --dport 68 -j RETURN
-A neutron-linuxbri-18bca5b86-2 -m set --match-set NIPV4a7eae0d2-74f8-4017-bf5d- src -j RETURN
-A neutron-linuxbri-18bca5b86-2 -m state --state INVALID -m comment --comment "Drop packets that appear relate
have an entry in conntrack." -j DROP
-A neutron-linuxbri-18bca5b86-2 -m comment --comment "Send unmatched traffic to the fallback chain." -j neutro
-A neutron-linuxbri-08bca5b86-2 -p udp -m udp --sport 68 -m udp --dport 67 -m comment --comment "Allow DHCP cl
-A neutron-linuxbri-08bca5b86-2 -j neutron-linuxbri-s8bca5b86-2
-A neutron-linuxbri-08bca5b86-2 -p udp -m udp --sport 67 -m udp --dport 68 -m comment --comment "Prevent DHCP
-A neutron-linuxbri-08bca5b86-2 -m state --state RELATED,ESTABLISHED -m comment --comment "Direct packets asso
BN
-A neutron-linuxbri-08bca5b86-2 -j RETURN
-A neutron-linuxbri-08bca5b86-2 -m state --state INVALID -m comment --comment "Drop packets that appear relate
have an entry in conntrack." -j DROP
-A neutron-linuxbri-08bca5b86-2 -m comment --comment "Send unmatched traffic to the fallback chain." -j neutro
-A neutron-linuxbri-s8bca5b86-2 -s 172.16.100.3/32 -m mac --mac-source FA:16:3E:2F:14:03 -m comment --comment
-A neutron-linuxbri-s8bca5b86-2 -m comment --comment "Drop traffic without an IP/MAC allow rule." -j DROP
-A neutron-linuxbri-sg-chain -m physdev --physdev-out tap8bca5b86-23 --physdev-is-bridged -m comment --comment
ca5b86-2
-A neutron-linuxbri-sg-chain -m physdev --physdev-in tap8bca5b86-23 --physdev-is-bridged -m comment --comment
asb86-2
-A neutron-linuxbri-sg-chain -j ACCEPT
-A neutron-linuxbri-sg-fallback -m comment --comment "Default drop rule for unmatched traffic." -j DROP
-A nova-api-INPUT -d 192.168.104.10/32 -p tcp -m tcp --dport 8775 -j ACCEPT
-A nova-filter-top -j nova-api-local
COMMIT
```

图 9-217

cirros-vm1 的 TAP interface 为 tap8bca5b86-23，可以看到：

(1) iptables 的规则是应用在 Neutron port 上的，port 在这里是 cirros-vm1 的虚拟网卡 tap8bca5b86-23。



- (2) ingress 规则集中定义在命名为“neutron-linuxbri-i8bca5b86-2”的 chain 中。
- (3) egress 规则集中定义在命名为“neutron-linuxbri-o8bca5b86-2”的 chain 中。

下面我们通过 dhcp namespace 对 cirros-vm1 进行 ping 和 ssh 测试，如图 9-218 所示。

```

root@devstack-controller:~# ip netns
qdhcp-7e95400c-d094-4de9-bb6d-b5d25effe362
root@devstack-controller:~#
root@devstack-controller:~# ip netns exec qdhcp-7e95400c-d094-4de9-bb6d-b5d25effe362 ping 172.16.100.3
PING 172.16.100.3 (172.16.100.3) 56(84) bytes of data.

^C
--- 172.16.100.3 ping statistics ---
48 packets transmitted, 0 received, 100% packet loss, time 47376ms
root@devstack-controller:~# ip netns exec qdhcp-7e95400c-d094-4de9-bb6d-b5d25effe362 ssh 172.16.100.3

^C
root@devstack-controller:~#

```

图 9-218

无法 ping 和 ssh cirros-vm1，可见当前的规则实现了“default”安全组，所有 ingress 流量都被禁止。

## 2. 应用新的安全组

本节我们将通过配置安全组让 cirros-vm1 能够被 ping 和 ssh。

有两种方法可以达到这个目的：

- (1) 修改“default”安全组。
- (2) 为 cirros-vm1 添加新的安全组。

这里我们采用第二种方法。

在安全组列表页面单击“Create Security Group”，如图 9-219 所示。

**Create Security Group**

Name \*

allow ping & ssh

Description

Allow Ping and SSH instances.

Description:

Security groups are sets of IP filter rules that are applied to the network settings for the VM. After the security group is created, you can add rules to the security group.

Cancel Create Security Group

图 9-219

为安全组命名并单击“Create Security Group”。新的安全组“allow ping & ssh”创建成功，如图 9-220 所示。

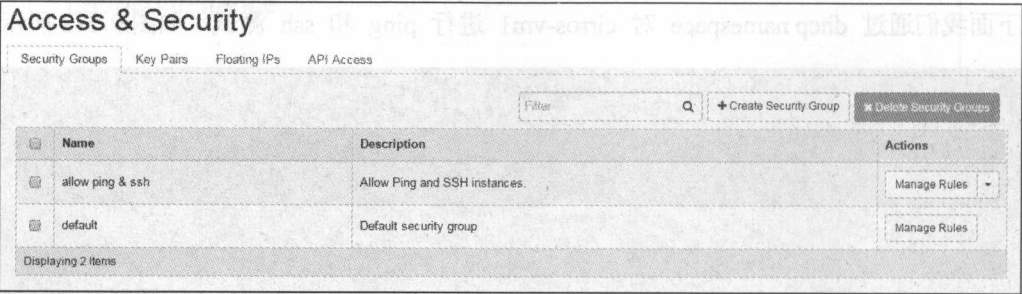


图 9-220

单击“Manage Rules”，查看“allow ping & ssh”的规则，如图 9-221 所示。

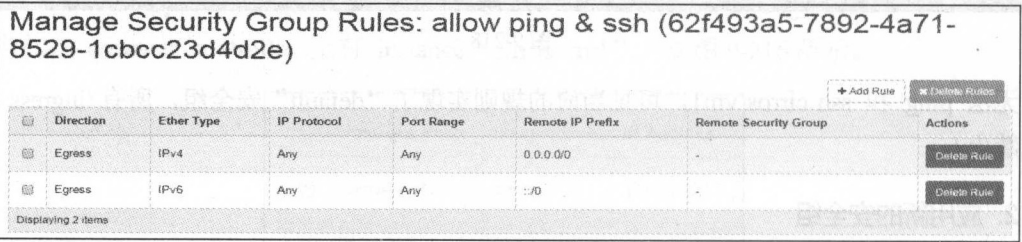


图 9-221

系统默认定义了两条规则，运行所有的外出流量。  
为清晰起见，可以单击“Delete Rule”删除这两条规则。  
单击“Add Rule”，添加允许 ping 的规则，如图 9-222 所示。

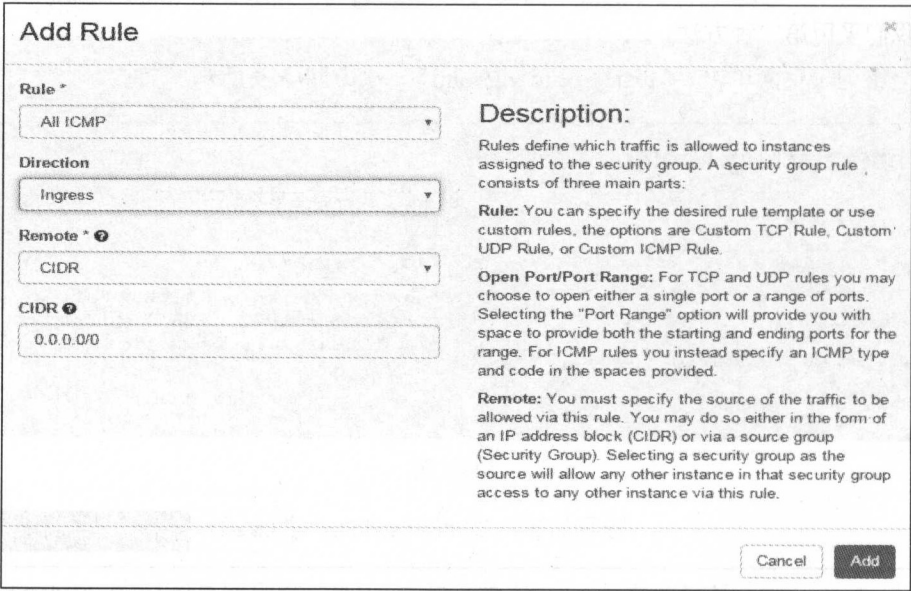
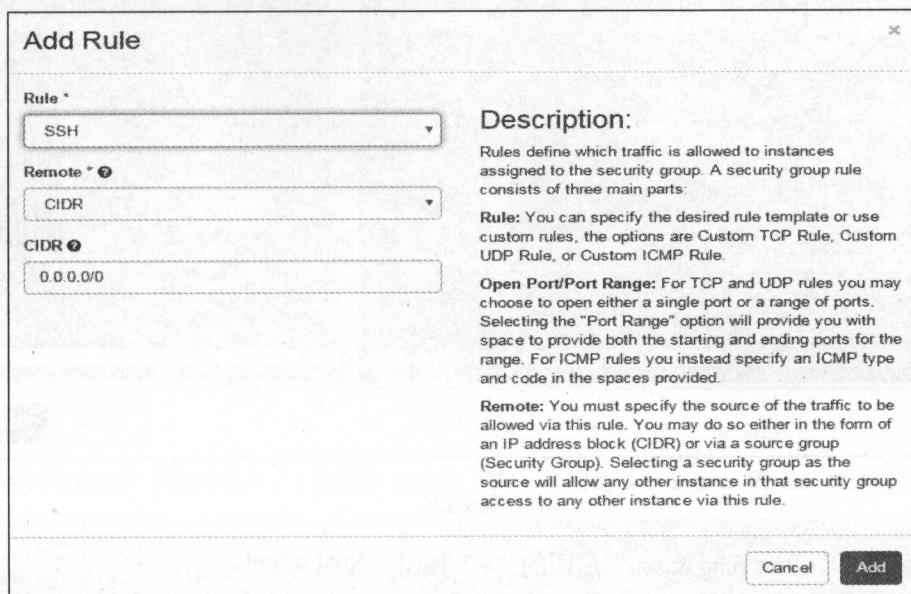


图 9-222

“Rule” 选择 “All ICMP”，“Direction” 选择 “Ingress”，然后单击 “Add”。同样的方式添加 ssh 规则，如图 9-223 所示。



**Add Rule**

Rule \*  
SSH

Remote \* ⓘ  
CIDR

CIDR ⓘ  
0.0.0.0/0

**Description:**  
Rules define which traffic is allowed to instances assigned to the security group. A security group rule consists of three main parts:  
**Rule:** You can specify the desired rule template or use custom rules, the options are Custom TCP Rule, Custom UDP Rule, or Custom ICMP Rule.  
**Open Port/Port Range:** For TCP and UDP rules you may choose to open either a single port or a range of ports. Selecting the "Port Range" option will provide you with space to provide both the starting and ending ports for the range. For ICMP rules you instead specify an ICMP type and code in the spaces provided.  
**Remote:** You must specify the source of the traffic to be allowed via this rule. You may do so either in the form of an IP address block (CIDR) or via a source group (Security Group). Selecting a security group as the source will allow any other instance in that security group access to any other instance via this rule.

Cancel Add

图 9-223

在列表中查看添加成功的规则，如图 9-224 所示。

Manage Security Group Rules: allow ping & ssh (62f493a5-7892-4a71-8529-1cbcc23d4d2e)

+ Add Rule -x Delete Rules

Direction	Ether Type	IP Protocol	Port Range	Remote IP Prefix	Remote Security Group	Actions
Ingress	IPv4	ICMP	Any	0.0.0.0/0	-	Delete Rule
Ingress	IPv4	TCP	22 (SSH)	0.0.0.0/0	-	Delete Rule

Displaying 2 items

图 9-224

接下来设置 cirros-vm1，使用新的安全组。

进入 instance 列表页面，单击 cirros-vm1 下拉操作列表中的 “Edit Security Groups”，如图 9-225 所示。

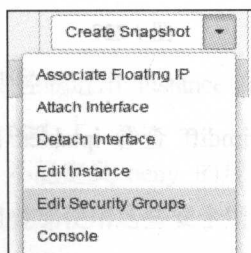


图 9-225

可以看到 cirros-vm1 当前使用的安全组为 “default”，可选安全组为 “allow ping & ssh”，

如图 9-226 所示。

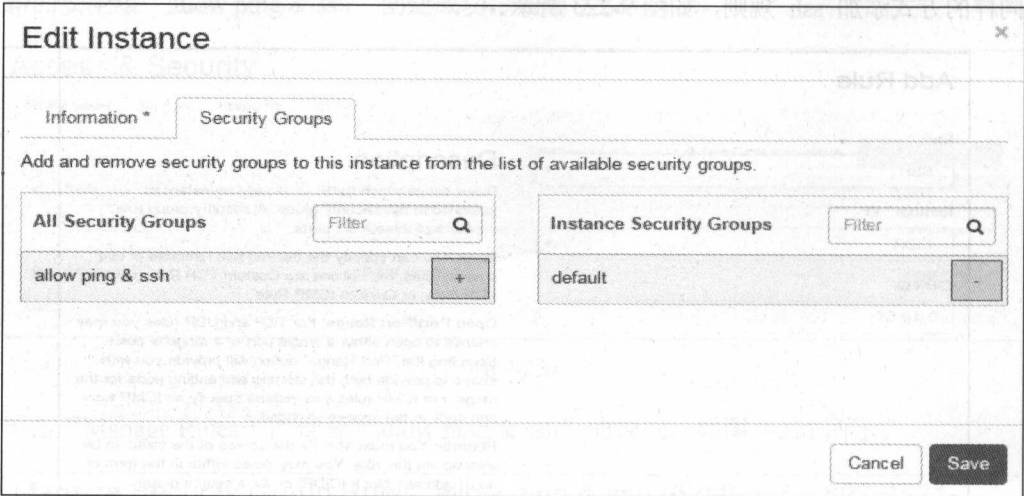


图 9-226

单击安全组 “allow ping & ssh” 后面的 “+” 按钮，如图 9-227 所示。

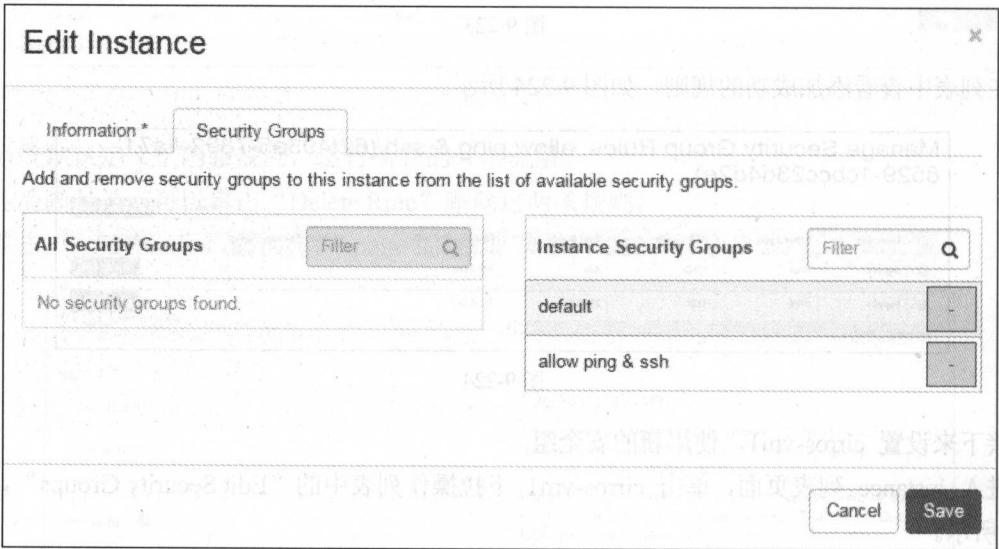


图 9-227

单击 “Save” 保存。

iptables 会立即更新，下面通过 vimdiff 查看 iptables 前后的变化，如图 9-228 所示。



```

root@devstack-controller: ~ x root@devstack-compute1: ~
:neutron-linuxbri-FORWARD - [0:0]
:neutron-linuxbri-INPUT - [0:0]
:neutron-linuxbri-OUTPUT - [0:0]
:neutron-linuxbri-i8bca5b86-2 - [0:0]
:neutron-linuxbri-local - [0:0]
+-- 29 lines: :neutron-linuxbri-i8bca5b86-2
-A neutron-filter-top -j neutron-linuxbri
-A neutron-linuxbri-FORWARD -m physdev -
-A neutron-linuxbri-FORWARD -m physdev --physdev-out tap8bca5b86-2
-A neutron-linuxbri-INPUT -m physdev --physdev-in tap8bca5b86-23
-A neutron-linuxbri-INPUT -m physdev --state RELATED,ESTABLISH
-A neutron-linuxbri-i8bca5b86-2 -s 172.16.100.2/32 -o udp -m udp
-A neutron-linuxbri-i8bca5b86-2 -p tcp -m tcp --dport 22 -j RETURN
-A neutron-linuxbri-i8bca5b86-2 -p icmp -j RETURN
-A neutron-linuxbri-i8bca5b86-2 -m state --state INVALID --comment
-A neutron-linuxbri-i8bca5b86-2 -m comment --comment
-A neutron-linuxbri-i8bca5b86-2 -p udp -m udp --sport 68 -m udp --
-A neutron-linuxbri-i8bca5b86-2 -j neutron-linuxbri-i8bca5b86-2
-A neutron-linuxbri-i8bca5b86-2 -p udp -m udp --sport 67 -m udp
+-- 7 lines: :neutron-linuxbri-i8bca5b86-2 -m state --state REL
-A neutron-linuxbri-sg-chain -m physdev --physdev-in tap8bca5b86-2
-A neutron-linuxbri-sg-chain -j ACCEPT
-A neutron-linuxbri-sg-fallback -m comment --comment
-A nova-api-INPUT -d 192.168.104.10/32 -p tcp -m tcp --dport 8775
-A nova-filter-top -j nova-api-local
c 135.1 97% d 137.1 97%

```

图 9-228

“allow ping & ssh”安全组引入了下面两条 iptables 规则，作用是运行 ingress 的 ssh 和 ping 流量。

```

-A neutron-linuxbri-i8bca5b86-2 -p tcp -m tcp --dport 22 -j RETURN
-A neutron-linuxbri-i8bca5b86-2 -p icmp -j RETURN

```

测试一下，现在能够 ping 和 ssh cirros-vm1 了，如图 9-229 所示。

```

root@devstack-controller:~
root@devstack-controller:~ # ip netns exec qdhcp-7e95400c-d094-4de9-bb6d-b5d25effe362 ping 172.16.100.3
PING 172.16.100.3 (172.16.100.3) 56(84) bytes of data.
64 bytes from 172.16.100.3: icmp_seq=1 ttl=64 time=1.10 ms
64 bytes from 172.16.100.3: icmp_seq=2 ttl=64 time=0.661 ms
64 bytes from 172.16.100.3: icmp_seq=3 ttl=64 time=0.615 ms
64 bytes from 172.16.100.3: icmp_seq=4 ttl=64 time=0.732 ms
64 bytes from 172.16.100.3: icmp_seq=5 ttl=64 time=0.588 ms
--- 172.16.100.3 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 3999ms
rtt min/avg/max/mdev = 0.588/0.740/1.108/0.193 ms
root@devstack-controller:~ # ip netns exec qdhcp-7e95400c-d094-4de9-bb6d-b5d25effe362 ssh cirros@172.16.100.3
The authenticity of host '172.16.100.3 (172.16.100.3)' can't be established.
RSA key fingerprint is 63:04:ec:71:c1:9f:b0:49:e2:b8:ea:af:3d:ab:19:2b.
Are you sure you want to continue connecting (yes/no)? yes
warning: Permanently added '172.16.100.3' (RSA) to the list of known hosts.
cirros@172.16.100.3's password:
$ ifconfig
eth0
Link encap:Ethernet HWaddr FA:16:3E:2F:14:03
inet addr:172.16.100.3 Bcast:172.16.100.255 Mask:255.255.255.0
inet6 addr: fe80::f816:3eff:fe2f:1403/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:288 errors:0 dropped:5 overruns:0 frame:0
TX packets:124 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:128650 (27.9 KiB) TX bytes:13354 (13.0 KiB)

```

图 9-229

### 3. 小结

安全组有以下特性：

- (1) 通过宿主机上 iptables 规则控制进出 instance 的流量。
- (2) 安全组作用在 instance 的 port 上。
- (3) 安全组的规则都是 allow，不能定义 deny 的规则。
- (4) instance 可应用多个安全组叠加使用这些安全组中的规则。

## 9.4.11 Firewall as a Service

### 1. 理解概念

Firewall as a Service (FWaaS) 是 Neutron 的一个高级服务。

让用户能够创建和管理防火墙，在 subnet 的边界上对 layer 3 和 layer 4 的流量进行过滤。传统网络中的防火墙一般放在网关上，用来隔离子网之间的访问。

FWaaS 的原理也一样，是在 Neutron 虚拟 router 上应用防火墙规则，控制进出租户网络的数据。

FWaaS 有三个重要概念：Firewall、Policy 和 Rule。

#### ● Firewall

租户能够创建和管理的逻辑防火墙资源。

Firewall 必须关联某个 Policy，因此必须先创建 Policy。

#### ● Firewall Policy

Policy 是 Rule 的集合，Firewall 会按照 Policy 中指定的顺序应用这些 Rule。

#### ● Firewall Rule

Rule 即访问控制的规则，包括源和目的 layer3 子网 IP、源和目的 layer4 端口、协议、allow 或 deny 动作等。

例如，我们创建一个 Rule，允许其他网络对内部子网中 instance 端口 22 的 TCP 请求。与 FWaaS 容易混淆的概念是安全组（Security Group）。

安全组的应用对象是虚拟网卡，由 L2 Agent 来实现，比如 neutron\_openvswitch\_agent 和 neutron\_linuxbridge\_agent，安全组会在计算节点上通过配置 iptables 规则来限制虚拟网卡的进出访问，即安全组保护的是 instance。

FWaaS 的应用对象是 router，可以在安全组之前隔离外部过来的恶意流量，但是对于同一个 subnet 内部不同虚拟网卡间的通信不作限制，即 FWaaS 保护的是 subnet。

所以可以同时部署 FWaaS 和安全组实现双重防护。

### 2. 启用 FWaaS

因为 FWaaS 是在 router 中实现的，所以 FWaaS 没有单独的 agent。

已有的 L3 Agent 负责提供所有 FWaaS 功能。

要启用 FWaaS，必须在 Neutron 的相关配置文件中做些设置。

#### (1) 配置 firewall driver

Neutron 在 /etc/neutron/fwaas\_driver.ini 文件中保存 FWaaS driver 的配置，如图 9-230 所示。

```
[fwaas]  
driver = neutron_fwaas.services.firewall.drivers.linux:iptables_fwaas.IptablesFwaasDriver  
enabled = True
```

图 9-230

driver 为 iptables。

以后如果支持新的 driver，可以在这里替换。

### (2) 配置 Neutron

除了配置 firewall driver，还必须在 Neutron 配置文件 /etc/neutron/neutron.conf 的 Service Plugins 中配置启用 FWaaS，如图 9-231 所示。

```
service_plugins = neutron.services.l3_router.l3_router_plugin.L3RouterPlugin,neutron.lbaas.services.loadbalancer.plugin.LoadBalancerPlugin,neutron_vpnaas.services.vpn.plugin.VPNDriverPlugin,  
neutron_fwaas.services.firewall.fwaas_plugin.FirewallPlugin
```

图 9-231

## 3. 实践 FWaaS

下面通过实验来学习 FWaaS。

在我们的实验环境中，有两个 instance：cirros-vm1（172.16.100.3）和 cirros-vm2（172.16.101.3），如图 9-232 所示。

cirros-vm1 和 cirros-vm2 分别位于网络 vlan100 和 vlan101。

vlan100 和 vlan101 之间由虚拟路由器 test\_router 连接。

网络拓扑如图 9-233 所示。

Instance Name	Image Name	IP Address
cirros-vm2	cirros	172.16.101.3
cirros-vm1	cirros	172.16.100.3
Displaying 2 items		

图 9-232

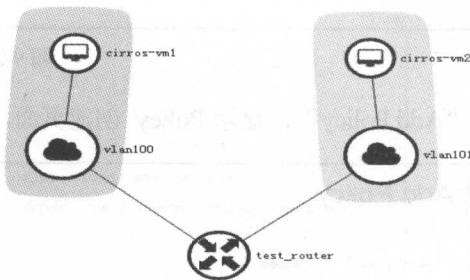


图 9-233

在 test\_router 没有应用任何 FWaaS 的情况下，cirros-vm1 可以通过 ping 和 ssh 跨网络访问 cirros-vm2，如图 9-234 所示。

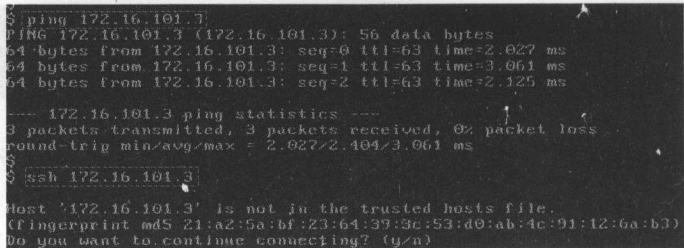


图 9-234

下面我们将进行如下实验：

创建一个不包含任何 rule 的 firewall “test\_firewall” 并应用到 test\_router。此时 FWaaS 生效，默认情况下会阻止任何跨子网的流量。

创建 rule 允许 ssh，并将其添加到 test\_firewall。此时 cirros-vm1 应该能够 ssh cirros-vm2。

(1) 应用无 rule 的 firewall

单击菜单 Project → Network → Firewalls，打开 Firewall Policies 标签页面。

目前没有定义任何 Policies，如图 9-235 所示。

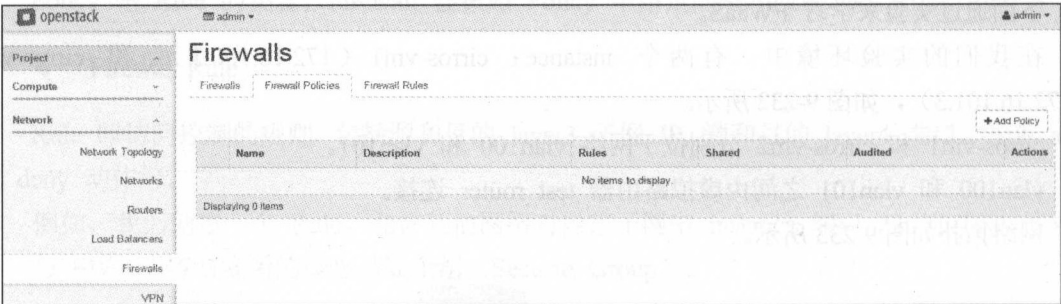


图 9-235

单击 “Add Policy”，显示 Policy 创建页面，如图 9-236 所示。

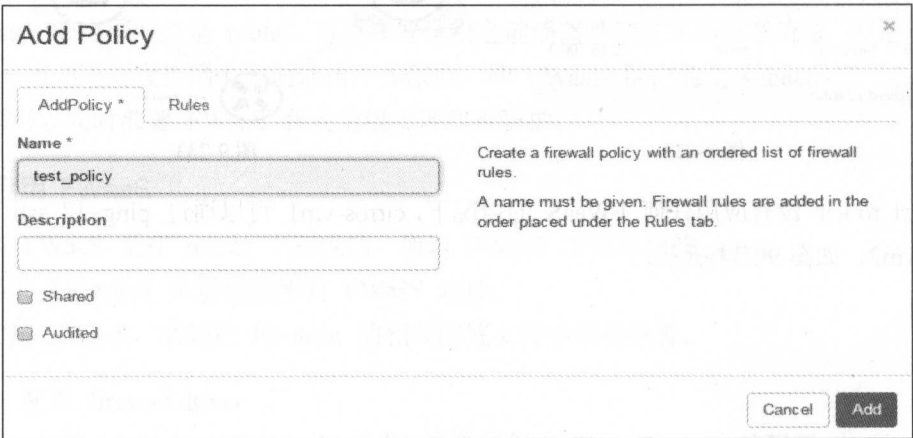


图 9-236



将 Policy 命名为“test\_policy”，直接单击“Add”，如图 9-237 所示。

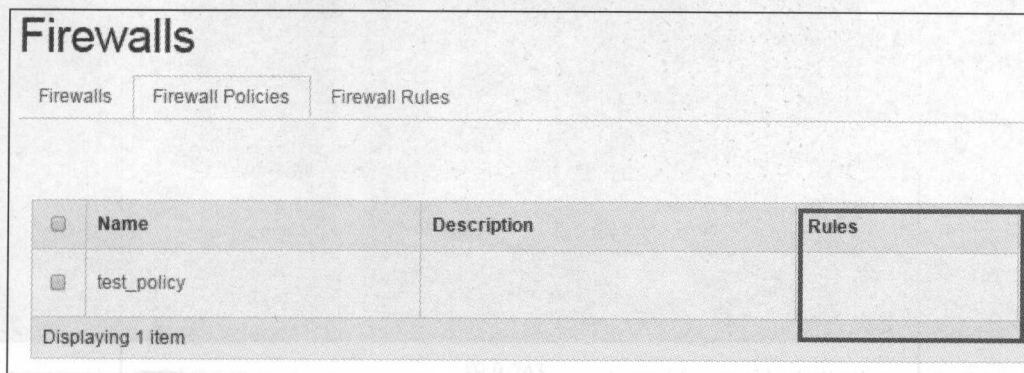


图 9-237

这样我们创建的 test\_policy 不包含任何 Rule。

进入“Firewalls”标签页，单击“Create Firewall”，如图 9-238 所示。

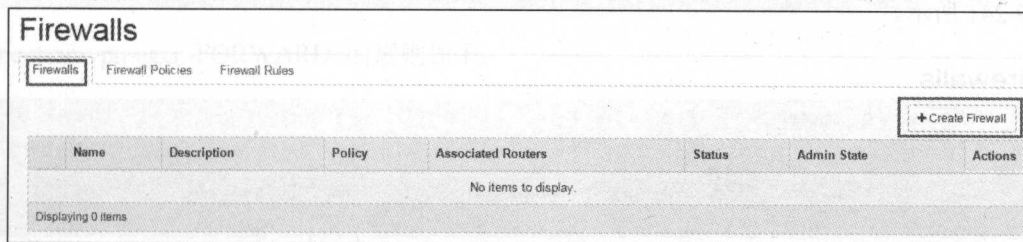


图 9-238

将新的 Firewall 命名为“test\_firewall”，并关联“test\_policy”，如图 9-239 所示。

The screenshot shows the 'Add Firewall' dialog box. It has two tabs: 'AddFirewall \*' and 'Routers'. The 'AddFirewall \*' tab is active. It contains the following fields:
 

- Name:** test\_firewall
- Description:** (empty)
- Policy \*:** test\_policy (selected from a dropdown)
- Admin State \*:** UP (selected from a dropdown)

 To the right of the fields, there is a note: 'Create a firewall based on a policy. A policy must be selected. Other fields are optional.' At the bottom right, there are 'Cancel' and 'Add' buttons.

图 9-239

在“Routers”标签页中选择“test\_router”。

单击“Add”，创建 firewall，如图 9-240 所示。

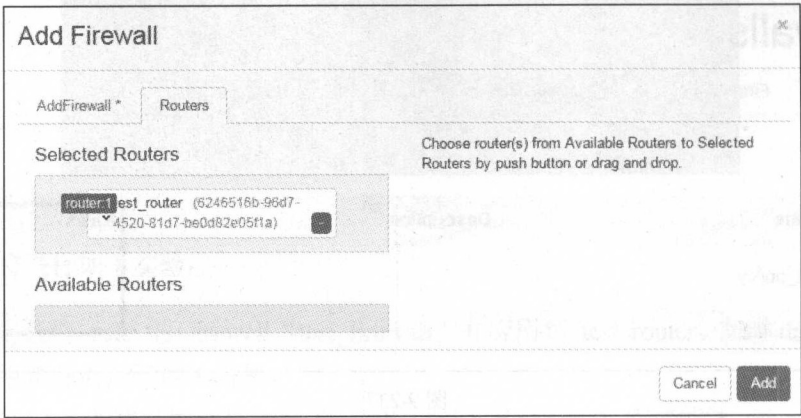


图 9-240

等待 test\_firewall 的 Status 变为“Active”，此时 test\_router 已经成功应用 test\_policy，如图 9-241 所示。

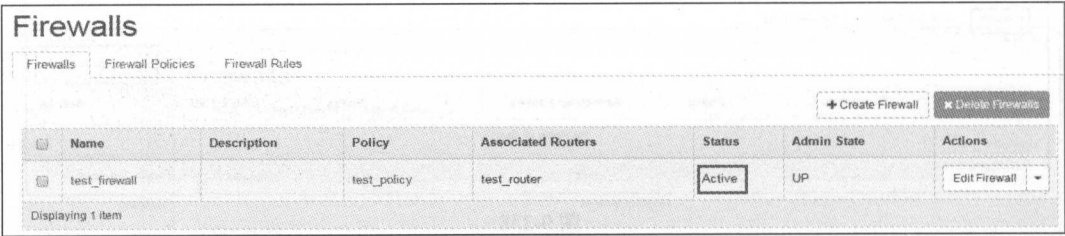


图 9-241

可以通过 iptables-save 查看 router namespace 的 iptables 规则，如图 9-242 所示。

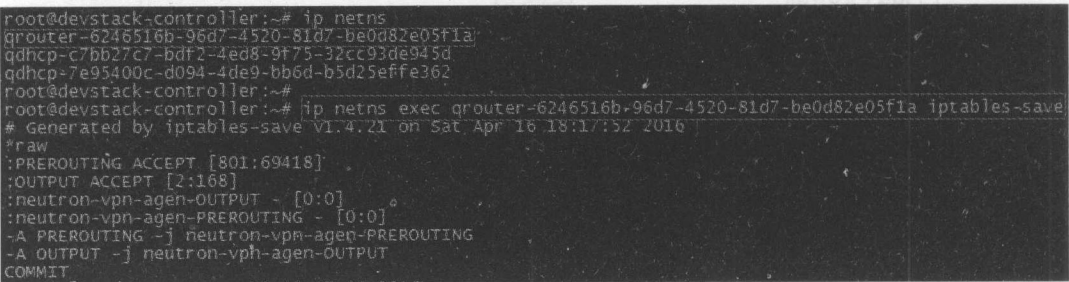


图 9-242

为了让大家了解底层到底发生了什么变化，下面用 vimdiff 显示了应用 test\_firewall 前后 iptables 规则的变化，如图 9-243 所示。

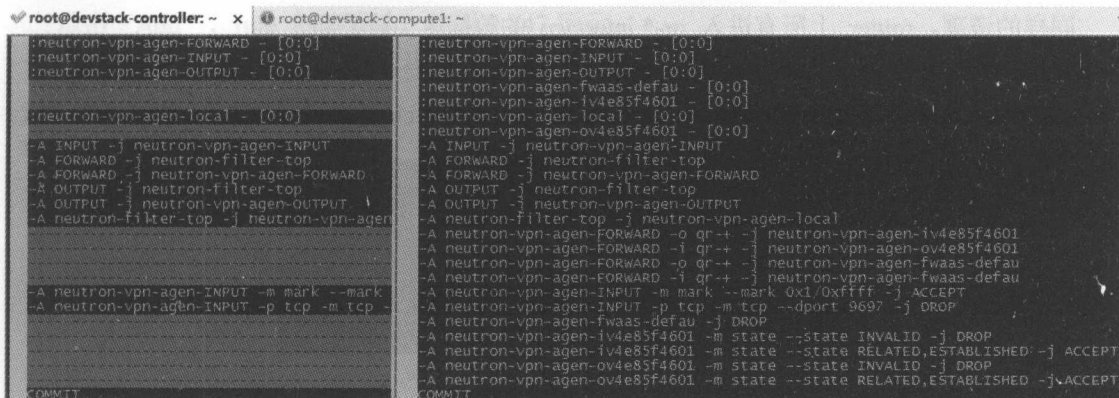


图 9-24

下面我们来分析一下这些规则。

route 在转发数据包时会使用 chain:

```
-A FORWARD -j neutron-vpn-agen-FORWARD
```

neutron-vpn-agen-FORWARD 的规则如下:

```
-A neutron-vpn-agen-FORWARD -o qr+ -j neutron-vpn-agen-iv4e85f4601
-A neutron-vpn-agen-FORWARD -i qr+ -j neutron-vpn-agen-ov4e85f4601
-A neutron-vpn-agen-FORWARD -o qr+ -j neutron-vpn-agen-fwaas-defau
-A neutron-vpn-agen-FORWARD -i qr+ -j neutron-vpn-agen-fwaas-defau
```

我们以第一条为例, 其含义是: 从 router namespace 任何一个 qr-\* interface 发出的流量都会应用 chain neutron-vpn-agen-iv4e85f4601, 该 chain 定义如下:

```
-A neutron-vpn-agen-iv4e85f4601 -m state --state INVALID -j DROP
-A neutron-vpn-agen-iv4e85f4601 -m state --state RELATED,ESTABLISHED -j ACCEPT
```

其规则为:

- 如果数据包的状态为 INVALID, 则 DROP。
- 如果数据包的状态为 RELATED 或 ESTABLISHED, 则 ACCEPT。

其他正常传输的数据怎么处理呢?

回到 neutron-vpn-agen-FORWARD chain 的下一条关于 router 外出数据的规则:

```
-A neutron-vpn-agen-FORWARD -o qr+ -j neutron-vpn-agen-fwaas-defau
```

neutron-vpn-agen-fwaas-defau 内容为:

```
-A neutron-vpn-agen-fwaas-defau -j DROP
```

可见, 数据会被丢弃。

同样的道理，router 上所有进入 qr-\* interface 的数据也会被丢弃。  
其结论是：在没有定义任何 firewall rule 的情况下，进出 router 的数据包都会被丢弃。  
ping 和 ssh 测试表明目前 cirros-vm1 确实已经无法与 cirros-vm2 通信，如图 9-244 所示。

```
$ ping 172.16.101.3
PING 172.16.101.3 (172.16.101.3): 56 data bytes
--- 172.16.101.3 ping statistics ---
7 packets transmitted, 0 packets received, 100% packet loss
$ ssh 172.16.101.3
$
```

图 9-244

(2) 允许 ssh

下面我们添加一条 firewall rule：允许 ssh。  
在 Firewall Rules 标签页面单击“Add Rule”，如图 9-245 所示。

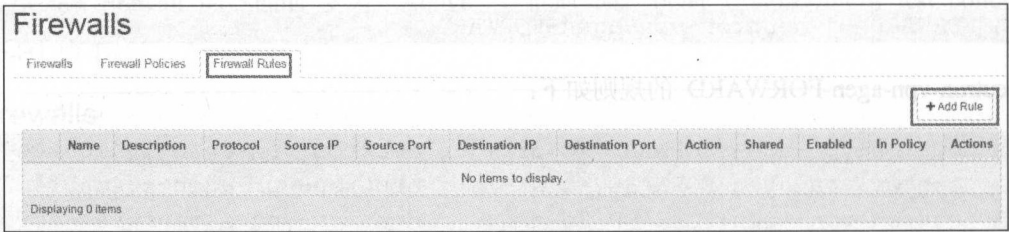


图 9-245

将新 rule 命名为“allow ssh”，Protocol 选择“TCP”，Action 为“ALLOW”，Destination Port/Port Range 为“22”，如图 9-246 所示。

Add Rule

AddRule\*

Name

allow ssh

Description

Protocol\*

TCP

Action\*

ALLOW

Source IP Address/Subnet

Destination IP Address/Subnet

Source Port/Port Range

Destination Port/Port Range

22

☒ Shared

☒ Enabled

Cancel

Add

Create a firewall rule.  
Protocol and action must be specified.  
Other fields are optional.

图 9-246



单击“Add”，rule 创建成功，如图 9-247 所示。

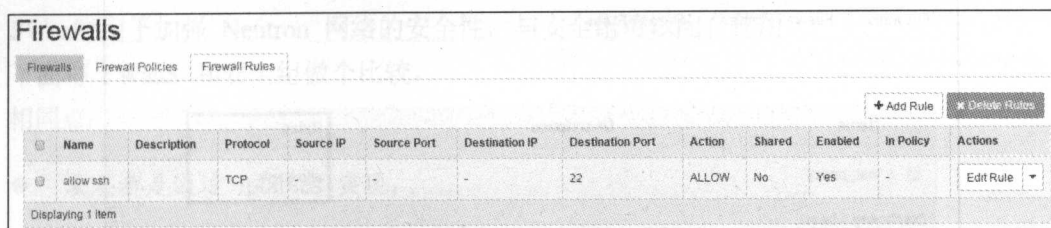


图 9-247

接下来将 rule 添加到 policy 中。

单击 Firewall Policies 标签页面，单击“test\_policy”后面的“Insert Rule”，如图 9-248 所示。

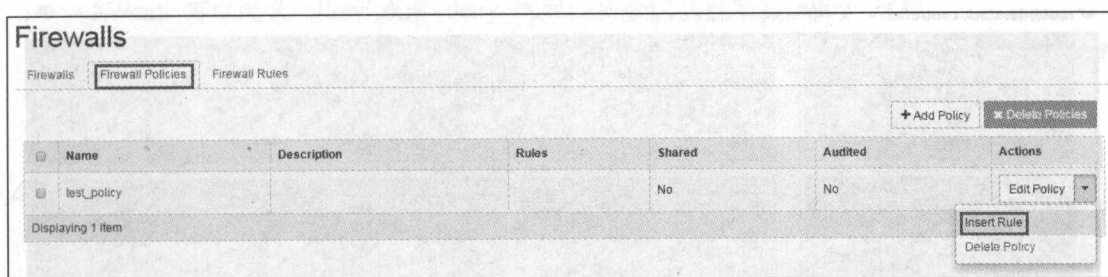


图 9-248

在 Insert Rule 下拉框中选择“allow ssh”，单击“Save Changes”，如图 9-249 所示。

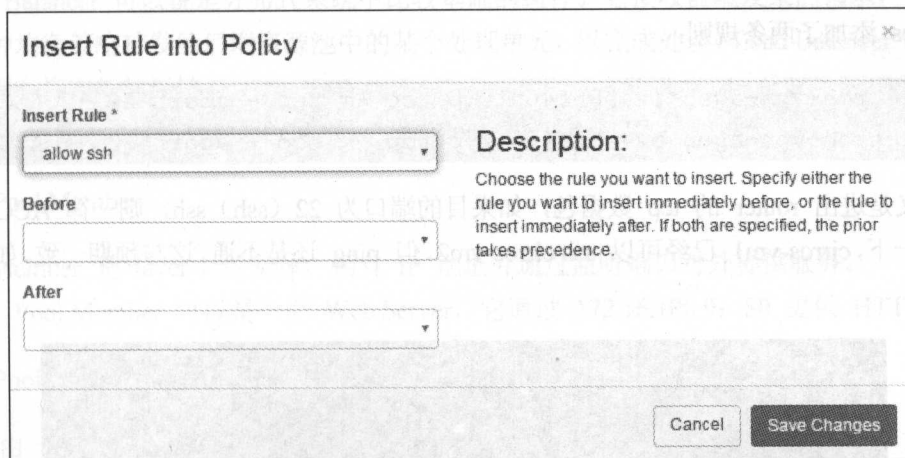


图 9-249

可以看到，“allow ssh”已经成功添加到“test\_policy”中，如图 9-250 所示。



#### 4. 小节

FWaaS 用于加强 Neutron 网络的安全性，与安全组可以配合使用。

下面将 FWaaS 和安全组做个比较。

相同点：

- 底层都是通过 iptables 实现。

不同点：

- FWaaS 的 iptables 规则应用在 router 上，保护整个租户网络；安全组则应用在虚拟网卡上，保护单个 instance。
- FWaaS 可以定义 allow 或者 deny 规则；安全组只能定义 allow 规则。
- 目前 FWaaS 规则不能区分进出流量，对双向流量都起作用；安全组规则可以区分 ingress 和 egress。

### 9.4.12 Load Balancing as a Service

#### 1. 理解概念

Load Balance as a Service (LBaaS) 是 Neutron 提供的一项高级网络服务。LBaaS 允许租户动态地在自己的网络中创建和管理 Load Balancer。

Load Balancer 可以说是分布式系统中比较基础的组件。它接收前端发来的请求，然后将请求按照某种均衡策略转发给后端资源池中的某个处理单元，以完成处理。Load Balancer 可以实现系统高可用和横向的扩展性。

LBaaS 有三个主要的概念：Pool Member、Pool 和 Virtual IP。

- Pool Member

Pool Member 是 layer 4 的实体，拥有 IP 地址并通过监听端口对外提供服务。

例如，Pool Member 可以是一个 Web Server，它通过 172.16.100.9: 80 提供 HTTP 服务。

- Pool

Pool 由一组 Pool Member 组成。

这些 Pool Member 通常提供同一类服务。

例如，有一个 web server pool，包含：

web1: 172.16.100.9: 80

web2: 172.16.100.10: 80

- Virtual IP

Virtual IP 也称作 VIP，是定义在 Load Balancer 上的 IP 地址。每个 Pool Member 都有自

己的 IP，但对外服务则是通过 VIP。Load Balancer 负责监听外部的连接，并将连接分发到 Pool Member。外部 Client 只知道 VIP，不知道也不需要关心是否有 Pool 或者有多少个 Pool Member。

OpenStack Neutron 目前默认通过 HAProxy 软件来实现 LBaaS。HAProxy 是一个流行的开源 Load Balancer。

Neutron 也支持其他一些第三方 Load Balancer。

如图 9-253 所示展示了 HAProxy 实现 Load Balancer 的方式。

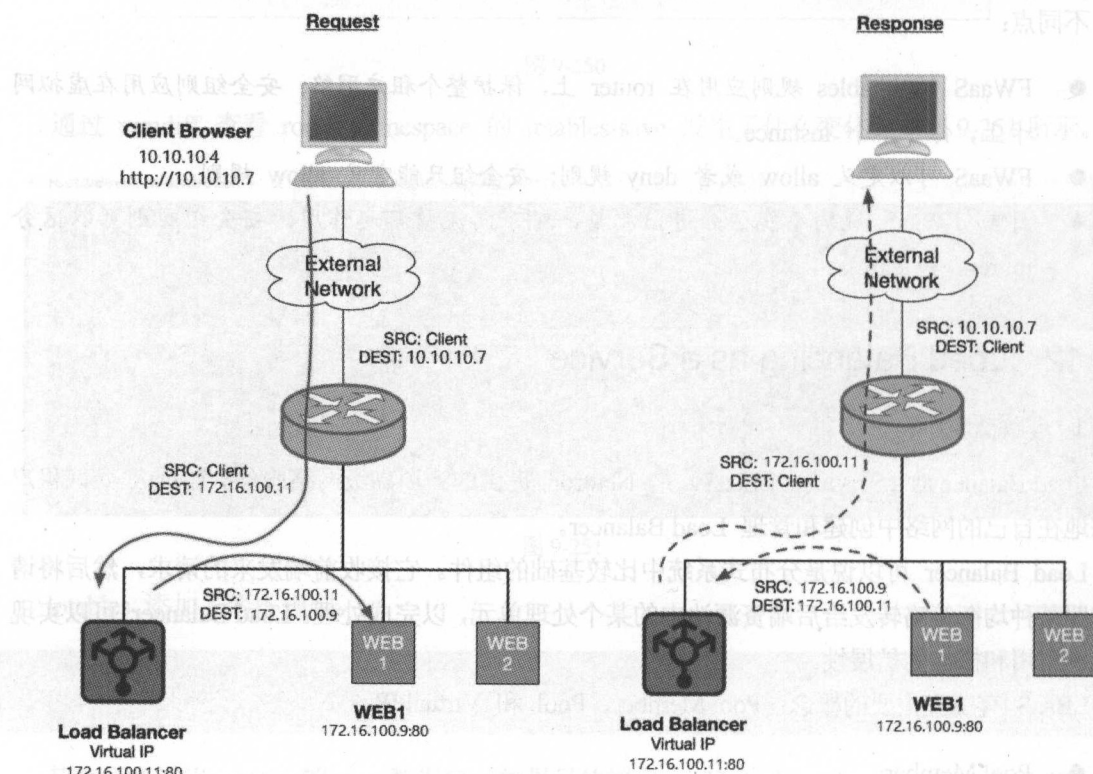


图 9-253

图 9-253 所示左图是 Client 发送请求到 Web Server 的数据流：

- (1) Load Balancer 接收到 Client 的请求。
- (2) Load Balancer 选择 Pool Member Web1，将数据包的目的 IP 设为 Web1 的地址 172.16.100.9。
- (3) 在将数据包转发给 Web1 之前，Load Balancer 将数据包的源 IP 修改为自己的 VIP 地址 172.16.100.11，其目的是保证 Web1 能够将应答数据发送回 Load Balancer。
- (4) 将数据包发送给 Web1。

图 9-262 所示右图是 Web Server 应答的数据流：

- (1) Web1 将数据包发送给 Load Balancer。



(2) Load Balancer 收到 Web1 发回的数据后, 将目的 IP 修改为 Client 的地址。

(3) Load Balancer 将数据包的源 IP 修改为 VIP 地址 172.16.100.11, 保证 Client 能够将后续的数据发送给自己。

(4) Load Balancer 将数据发送给 Client。

## 2. 启用 LBaaS

Neutron 通过 lbaas plugin 和 lbaas agent 提供 LBaaS 服务, 如图 9-254 所示。

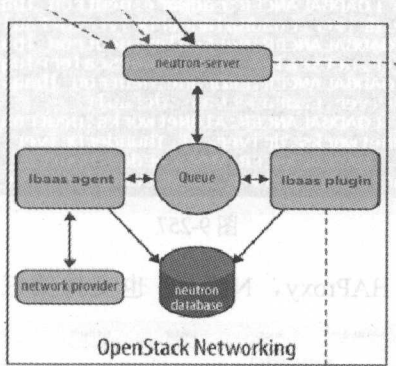


图 9-254

lbaas plugin 与 Neutron Server 一起运行在控制节点上。

lbaas agent 运行在网络节点上。

对于我们的实验环境, 控制节点和网络节点是一个, 都是 devstack-controller。

### (1) 配置 LBaaS agent

Neutron 存放 LBaaS agent 配置的文件是 /etc/neutron/services/loadbalancer/haproxy/lbaas\_agent.ini, 如图 9-255 所示。

```
[DEFAULT]
interface_driver = neutron.agent.linux.interface.BridgeInterfaceDriver
```

图 9-255

interface\_driver 设置 load balancer 的网络接口驱动, 可以有两个选项:

- Linux Bridge

```
interface_driver=neutron.agent.linux.interface.BridgeInterfaceDriver
```

- Open vSwitch

```
interface_driver=neutron.agent.linux.interface.OVSInterfaceDriver
```

### (2) 配置 LBaaS plugin

在 /etc/neutron/neutron.conf 中设置启用 LBaaS plugin, 如图 9-256 所示。

```
[DEFAULT]
service_plugins = neutron.services.l3_router.l3_router_plugin.L3RouterPlugin,neutron_lbaas.services.loadbalancer.plugin.LoadBalancerPlugin,neutron_vpnaas.services.vpn.plugin.VPNDriverPlugin,neutron_fwaas.services.firewall.plugin.fwaas_plugin.FirewallPlugin
```

图 9-256

在 `/etc/neutron/neutron_lbaas.conf` 中设置 service provider，如图 9-257 所示。

```
service_provider=LOADBALANCER:Haproxy:neutron_lbaas.services.loadbalancer.drivers.haproxy.plugin_driver.HaproxyOnHostPluginDriver:default
# service_provider = LOADBALANCER:radware:neutron_lbaas.services.loadbalancer.drivers.radware.driver.LoadBalancerDriver:default
# service_provider=LOADBALANCER:NetScaler:neutron_lbaas.services.loadbalancer.drivers.netscaler.netscaler_driver.NetscalerPluginDriver
# service_provider=LOADBALANCER:Embrane:neutron_lbaas.services.loadbalancer.drivers.embrane.driver.EmbraneLbaas:default
# service_provider = LOADBALANCER:A10Networks:neutron_lbaas.services.loadbalancer.drivers.a10networks.driver_v1.ThunderDriver:default
# service_provider = LOADBALANCER:VMwareEdge:neutron_lbaas.services.loadbalancer.drivers.vmware.edge_driver.EdgeLoadBalancerDriver:default
```

图 9-257

可以看到，除了默认的 HAProxy，Neutron 也支持第三方 provider，比如 radware，VMwareEdge 等。

重启 neutron 服务，确保 LBaaS 正常运行，如图 9-258 所示。

```
root@devstack-controller:~# ps -elf|grep*neutron-lbaas-agent
0 S root      3414  3234  1  80   0 - 31990 ep_pol Apr15 pts/14   00:56:34 /usr/bin/python /usr/local/bin/neutron-lbaas-agent --config-file /etc/neutron/neutron.conf --config-file=/etc/neutron/services/loadbalancer/haproxy/lbaas_agent.ini
```

图 9-258

### 3. 实践 LBaaS

本节我们将实践如下 LBaaS 环境，如图 9-259 所示。

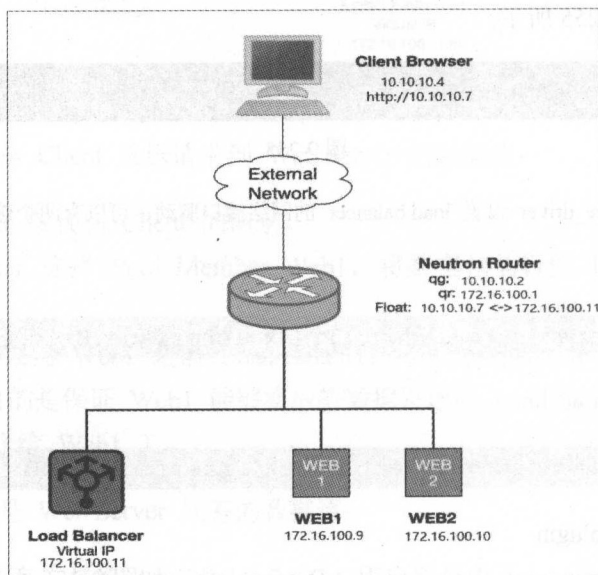


图 9-259

该环境描述如下：

- (1) 创建一个 Pool “web servers”。
- (2) 两个 Pool Member “WEB1” 和 “WEB2”，均为运行 Ubuntu cloud image 的 instance。
- (3) Load Balancer VIP 与 floating IP 关联。
- (4) 位于外网的 Client 通过 floating IP 外网访问 Web Server

下面我们将一步步实践 LBaaS。

4. 创建 Pool

单击菜单 Project → Network → Load Balancers，单击 Pools 标签页中的 “Add Pool”，如图 9-260 所示。

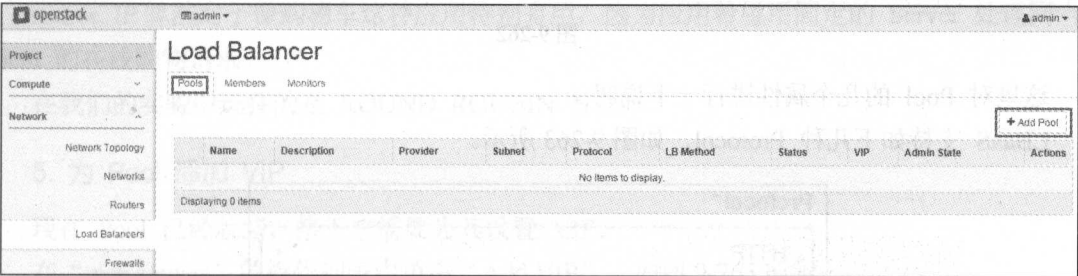


图 9-260

显示 Pool 创建页面，如图 9-261 所示。

The screenshot shows the 'Add Pool' form. At the top is a title 'Add Pool' and a close button. Below is a text input field for 'Add New Pool \*'. The form is divided into two columns. The left column contains several labeled input fields: 'Name \*' (with 'web servers' entered), 'Description' (empty), 'Provider' (a dropdown menu showing 'haproxy (default)'), 'Subnet \*' (a dropdown menu showing 'subnet\_172\_16\_100\_0 (172.16.100.0/24)'), 'Protocol \*' (a dropdown menu showing 'HTTP'), 'Load Balancing Method \*' (a dropdown menu showing 'ROUND\_ROBIN'), and 'Admin State \*' (a dropdown menu showing 'UP'). The right column contains explanatory text for the 'Load Balancing Method' section, detailing 'Round robin', 'Source IP', and 'Least connections' methods. At the bottom right of the form are 'Cancel' and 'Add' buttons.

图 9-261

将 Pool 命名为“web servers”。

Provider 选择默认的“haproxy”。

Subnet 选择“172.16.100.0/24”。

Protocol 选择“HTTP”。

Load Balancing Method 选择“ROUND\_ROBIN”。

单击“Add”，“web servers”创建成功，如图 9-262 所示。

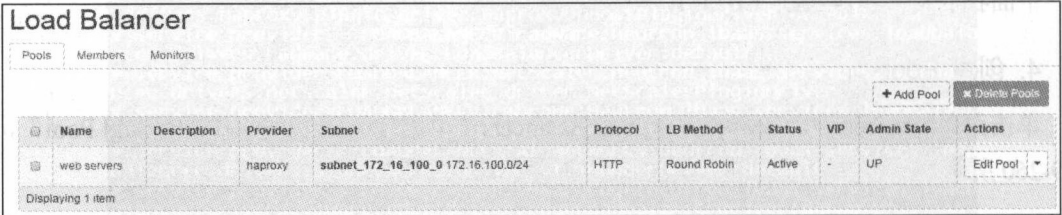


图 9-262

这里对 Pool 的几个属性进行一下说明。

LBaaS 支持如下几种 Protocol，如图 9-263 所示。

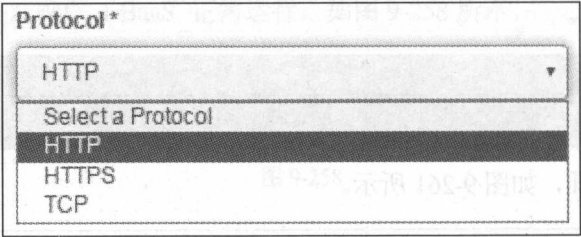


图 9-263

因为我们用 Web Server 做实验，所以这里需要选择“HTTP”。

LBaaS 支持多种 Load Balance Method，如图 9-264 所示。

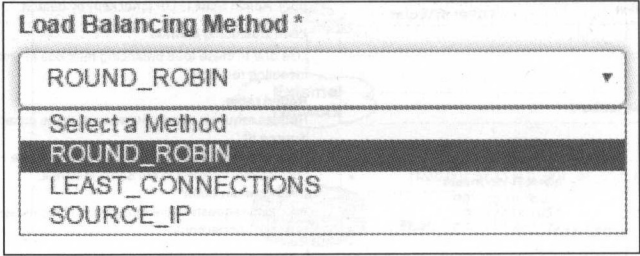


图 9-264

● ROUND\_ROUBIN

如果采用 round robin 算法，Load Balancer 按固定的顺序从 pool 中选择 member 相应 client 的连接请求。

这种方法的不足是缺乏机制检查 member 是否负载过重。



有可能出现某些 member 由于处理能力弱而不得不继续处理新连接的情况。

如果所有 Pool Member 具有相同处理能力、内存容量，并且每个连接持续的时间大致相同，那么将非常适合使用 round robin。因为这种情况下每个 member 的负载会非常均衡。

#### ● LEAST\_CONNECTIONS

如果采用 least connections 算法，Load Balancer 会挑选当前连接数最少的 Pool Member。

这是一种动态的算法，需要实时监控每个 member 的连接数量和状态。

处理能力强的 member 由于能够更快地处理连接，通常会分配到更多的新连接。

#### ● SOURCE\_IP

如果采用 source IP 算法，具有相同 source IP 地址的连接会被分发到同一个 Pool Member。

source IP 算法对于像购物车这种应用特别有用，因为应用希望用固定的 Server 处理同一个 Client 的在线购物请求。

在我们的实验中选择的是 ROUND\_ROUBIN 算法。

### 5. 为 Pool 添加 VIP

现在 Pool 已经就绪，接下来需要为其设置 VIP。

在“web servers”的操作列表中单击“Add VIP”，如图 9-265 所示。

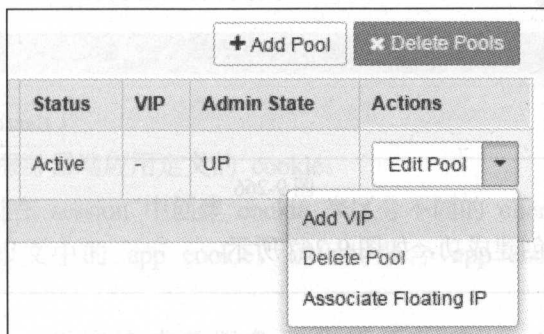


图 9-265

VIP 命名为“VIP for web servers”。

VIP Subnet 选择“172.16.100.0/24”，与 pool 一致。

指定 VIP 为 172.16.100.11，如果不指定，系统会自动从 subnet 中分配。

指定 HTTP 端口 80。

Session Persistence 选择“SOURCE IP”。

可以通过 Connection Limit 限制连接的数量，如果不填则为不加限制，如图 9-266 所示。

Add VIP

Specify VIP \*

Name \*

VIP for web servers

Description

VIP Subnet

subnet 172.16.100.0 (172.16.100.0)

IP address

172.16.100.11

Protocol Port \* ?

80

Protocol \*

HTTP

Session Persistence

SOURCE\_IP

Connection Limit ?

Admin State \*

UP

Create a VIP for this pool. Assign a name, description, IP address, port, and maximum connections allowed for the VIP. Choose the protocol and session persistence method for the VIP. Admin State is UP (checked) by default.

When no IP address is provided, the VIP will obtain an address from the selected subnet. If a specific IP address is desired, it may be provided and must also be an address within the selected subnet.

Cancel

Add

图 9-266

单击“Add”，VIP 创建成功，如图 9-267 所示。

Load Balancer

Pools

Members

Monitors

+ Add Pool

✖ Delete Pools

Name	Description	Provider	Subnet	Protocol	LB Method	Status	VIP	Admin State	Actions
web servers		haproxy	subnet_172_16_100_0 172.16.100.0/24	HTTP	Round Robin	Active	VIP for web servers Address: 172.16.100.11	UP	Edit Pool

Displaying 1 item

图 9-267

通常我们希望让同一个 Server 来处理某个 Client 的连续请求。否则 Client 可能会由于丢失 session 而不得不重新登录。这个特性就是 Session Persistence。

VIP 支持如下几种 Session Persistence 方式，如图 9-268 所示。

Session Persistence

SOURCE_IP
No Session Persistence
SOURCE_IP
HTTP_COOKIE
APP_COOKIE

图 9-268

### ● SOURCE\_IP

这种方式与前面 Load Balance 的 SOURCE\_IP 效果一样。

初始连接建立后, 后续来自相同 source IP 的 Client 请求会发送给相同的 member。

当大量 Client 通过同一个代理服务器访问 VIP 时, SOURCE\_IP 会造成 member 负载不均。

### ● HTTP\_COOKIE

HTTP\_COOKIE 的工作方式如下:

当 Client 第一次连接到 VIP 时, HAProxy 从 pool 中挑选出一个 member。

当此 member 响应请求时, HAProxy 会在应答报文中注入命名为“SRV”的 cookie, 这个 cookie 包含了该 member 的唯一标识。

client 的后续请求都会包含这个“SRV”cookie。

HAProxy 会分析 cookie 的内容, 并将请求转发给同一个 member。

HTTP\_COOKIE 优于 SOURCE\_IP, 因为它不依赖 client 数据包的 IP 地址。

### ● APP\_COOKIE

app cookie 依赖于服务器端应用定义的 cookie。

比如 app 可以通过在 session 中创建 cookie 来区分不同的 client。

HAProxy 会查看报文中的 app cookie, 确保将包含 app cookie 的请求发送到同一个 member。

如果没有 cookie (新连接或者服务器应用不创建 cookie), HAProxy 会采用 ROUND\_ROUBIN 算法分配 member。

区分 Load Balance Method 和 Session Persistence

前面我们介绍了三种 Load Balance Method, 如图 9-269 所示。

Load Balancing Method \*

ROUND_ROBIN
Select a Method
ROUND_ROBIN
LEAST_CONNECTIONS
SOURCE_IP

图 9-269

这里还有三种 Session Persistence, 如图 9-270 所示。

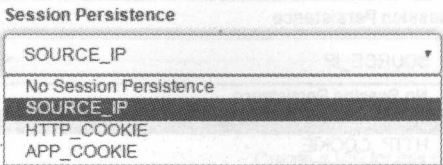


图 9-270

因为两者都涉及到如何选择 Pool Member，所以很容易混淆。它们之间的最大区别在于选择 Pool Member 的阶段不同：

- Load Balance Method 是为新连接选择 member 的方法。
- Session Persistence 是为同一个 Client 的后续连接选择 member 的方法。

例如这里我们的设置为：

```
Load Balance Method - ROUND_ROUBIN
Session Persistence - SOURCE_IP
```

当 Client A 向 VIP 发送第一个请求时，HAProxy 通过 ROUNDROUBIN 选择 member1。对于 Client A 后续的请求，HAProxy 则会应用 SOURCE\_IP 机制，仍然选择 member1 来处理请求。

6. 添加 Pool Member

现在我们已经有了 Pool “web servers”，接下来需要往 Pool 里添加 member。CloudMan 准备了两个 instance：“Web1”和“Web2”。

(1) 使用 Ubuntu Cloud Image

因为 cirros 镜像不能运行 HTTP 服务，所以使用的是 Ubuntu Cloud Image，如图 9-271 所示。Ubuntu Cloud Image 下载地址为 <http://uec-images.ubuntu.com/trusty/current/trusty-server-cloudimg-amd64-disk1.img>。

Instances						
		Instance Name		Filter		
	Instance Name	Image Name	IP Address	Size	Key Pair	Status
	Web2	Ubuntu 14.04 LTS Cloud Image	172.16.100.10	m1.tiny	cloud	Active
	Web1	Ubuntu 14.04 LTS Cloud Image	172.16.100.9	m1.tiny	cloud	Active
Displaying 2 items						

图 9-271

与以前的 instance 不同，Web1 和 Web2 使用了 Key Pair “cloud”，这是因为 Ubuntu Cloud Image 的 ubuntu 用户的密码是随机的，通过 Key Pair 访问是一个比较方便的方法。



下面是 Key Pair 的配置方法：

通过 ssh-keygen 生成 Key Pair，如图 9-272 所示。

```

root@devstack-compute1:~#
root@devstack-compute1:~# ssh-keygen -t rsa -f cloud.key
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in cloud.key.
Your public key has been saved in cloud.key.pub.
The key fingerprint is:
57:8d:0b:4e:e2:20:42:86:d8:b5:4c:c5:9a:e1:dd:f1 root@devstack-compute1
The key's randomart image is:
+--[ RSA 2048 ]-----+
|.oo.oo.|
|oo.o... ..o|
|..+=...oo o|
|..+.o.+Eo |
|  S o |
+-----+
root@devstack-compute1:~# ls -l cloud.key*
-rw----- 1 root root 1679 Apr 18 11:06 cloud.key
-rw-r--r-- 1 root root 404 Apr 18 11:06 cloud.key.pub
root@devstack-compute1:~#

```

图 9-272

其中 cloud.key.pub 是公钥，需要添加到 instance 的 ~/.ssh/authorized\_keys 文件中。

cloud.key 是私钥，用于访问 instance。

将 Pair Key 导入 OpenStack。

进入 Project → Compute → Access & Security 菜单，单击 Key Pairs 标签页的“Import Key Pair”，如图 9-273 所示。

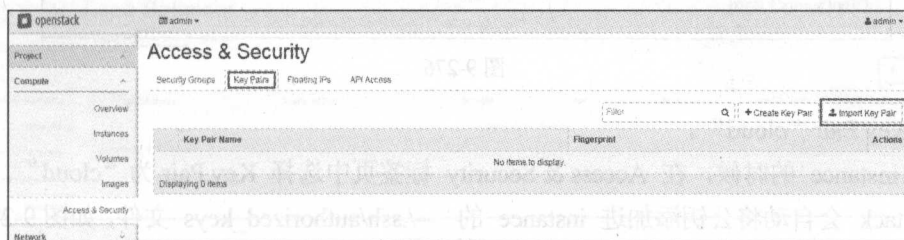


图 9-273

复制 cloud.key.pub 的内容，如图 9-274 所示。

```

root@devstack-compute1:~#
root@devstack-compute1:~# cat cloud.key.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQACI2DVu60IHfBSwKoswYElXEQARlqoSdLswj
C9/EhFIsw7ExeRn5F3fusbHwrj/1QhUdEWda1qcD+Dp63ACTkj5TEdbwtNufRrOwvQe/ubdQr
ysbkAvmh6p6S8QG9sFZdx6SKG+zKSMUUYKI044+kyzdbl awu6YjYqW6ubZLBGKSQuUTROuvOT
B5vp1lLlY4pwCz22k6BAthv3+q3yhXu6rs4pegLNb2p71RJ+w1lFIJZwM3Y1YPyRXNwWazwE
H+NhcWhqOURDb6HdruE88q0lvmmuU8ktvfTxMiu3ppRojaug5b7gKnPvr7wF9X0zocXzt5Bxp
AUDvqMdpTiwIwIX root@devstack-compute1
root@devstack-compute1:~#

```

图 9-274

粘贴到 Public Key 输入框中，给 Key Pair 命名为“cloud”，并单击“Import Key Pair”，如图 9-275 所示。

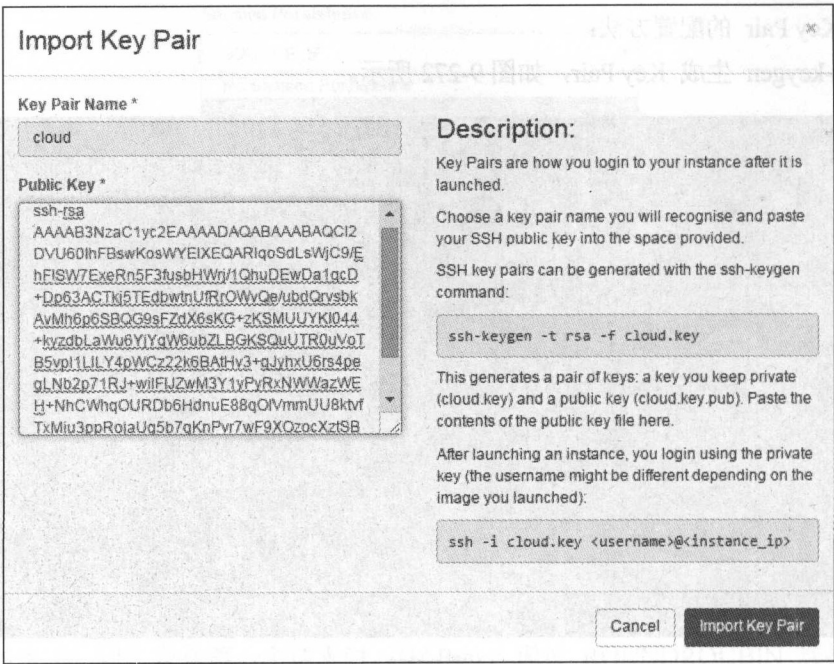


图 9-275

Key Pair “cloud” 成功导入，如图 9-276 所示。

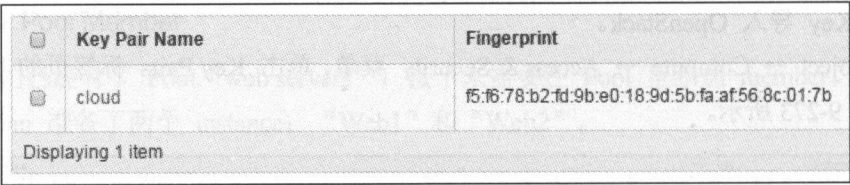


图 9-276

使用 Key Pair “cloud”。

launch instance 的时候，在 Access & Security 标签页中选择 Key Pair 为 “cloud”。

OpenStack 会自动将公钥添加进 instance 的 ~/.ssh/authorized\_keys 文件，如图 9-277 所示。

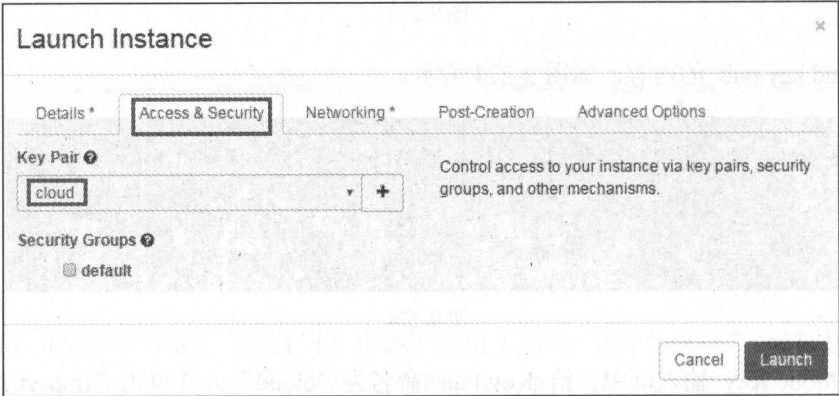


图 9-277

ssh instance。

用 `-i cloud.key` 指定私钥，并以 `ubuntu` 用户 `ssh` “Web1” 和 “Web2”，如图 9-278 所示。

```

root@devstack-controller:~#
root@devstack-controller:~# ip netns exec grouper-6246516b-96d7-4520-81d7
-be0d82e05fia/ssh -i cloud.key ubuntu@172.16.100.9
welcome to ubuntu 14.04.4 LTS (GNU/Linux 3.13.0-85-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

System information as of Sun Apr 17 08:58:41 UTC 2016

system load:  0.04      Processes:          73
usage of /:   39.9% of 2.13GB  Users logged in:   1
Memory usage: 14%      IP address for eth0: 172.16.100.9
Swap usage:   0%

Graph this data and manage this system at:
https://landscape.canonical.com/

Get cloud support with Ubuntu Advantage Cloud Guest:
http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.

Last login: Sun Apr 17 08:58:50 2016 from 172.16.100.1
ubuntu@web1:~$

```

图 9-278

无须密码直接登录 instance。

“Web1” 和 “Web2” 准备就绪了，可以添加到 Pool 中了。

进入 Project → Network → Load Balancers 的 Members 标签页中，单击 “Add Member”，如图 9-279 所示。

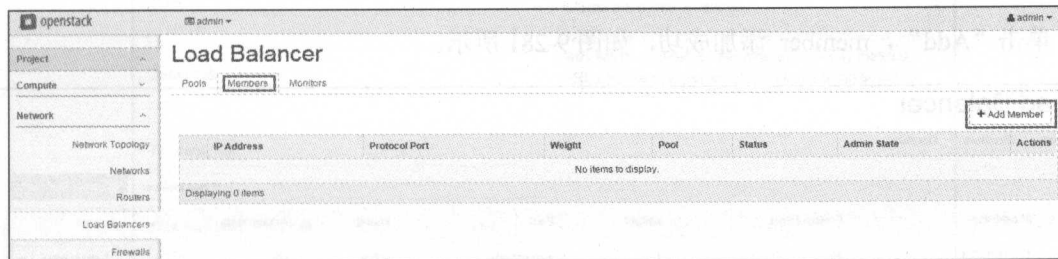


图 9-279

打开如图 9-280 所示的界面，配置如下：

- Pool 选择 “web servers”。
- Member(s) 中多选 “Web1” 和 “Web2”。
- Protocol Port 设置为 “80”。

Add Member

Add New Member \*

Pool \*

web servers

Member Source

Select from active instances

Member(s) ?

Web1  
Web2

Weight ?

Protocol Port \* ?

80

Admin State \*

UP

Add member(s) to the selected pool.  
Choose one or more listed instances to be added to the pool as member(s). Assign a numeric weight and port number for the selected member(s) to operate(s) on; e.g., 80.  
Only one port can be associated with each instance.

Cancel

Add

图 9-280

单击 “Add”，member 添加成功，如图 9-281 所示。

Load Balancer

Pools

Members

Monitors

+ Add Member

✖ Delete Members

	IP Address	Protocol Port	Weight	Pool	Status	Admin State	Actions
	172.16.100.9	80	1	web servers	Active	UP	<div>Edit Member</div>
	172.16.100.10	80	1	web servers	Active	UP	<div>Edit Member</div>

Displaying 2 items

图 9-281

7. 创建 Monitor

LBaaS 可以创建 monitor，用于监控 Pool Member 健康状态。

如果某个 member 不能正常工作，monitor 会将其状态设置为 down，从而避免将后续请求转发给它。

下面我们为 Pool 添加一个 monitor。

在 Monitors 标签页中单击 “Add Monitor”，如图 9-282 所示。



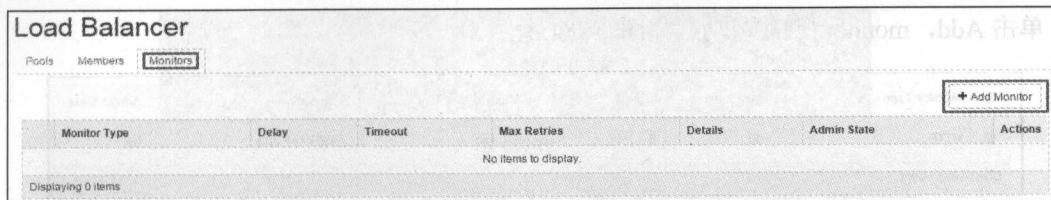


图 9-282

打开如图 9-283 所示的界面，配置如下：

- Type 选择“HTTP”，含义是通过 HTTP 检查 member 的健康状态。
- Delay 设置为“10”，含义是 10 秒检查一次 member 的状态。
- Timeout 设置为“5”，含义是如果 member 在 5 秒内无法应答，则超时。
- Max Retries 设置为“3”，含义是如果尝试 3 次都超时或者失败，则将 member 状态设置为 down。
- HTTP Method 设置为“GET”。
- URL 设置为“/”。
- Expected HTTP Status Codes 设置为“200”。

最后三项的含义是通过 HTTP GET 请求 member “/” URL，如果返回码为 200，则认为 member 状态正常。

图 9-283

单击 Add, monitor 创建成功, 如图 9-284 所示。。

Monitor Type	Delay	Timeout	Max Retries	Details	Admin State
HTTP	10	5	3	GET / => 200	UP

Displaying 1 item

图 9-284

下面将新建的 monitor 添加到 pool 。

在 “web servers” 的操作列表中单击 “Associate Monitor”，如图 9-285 所示。

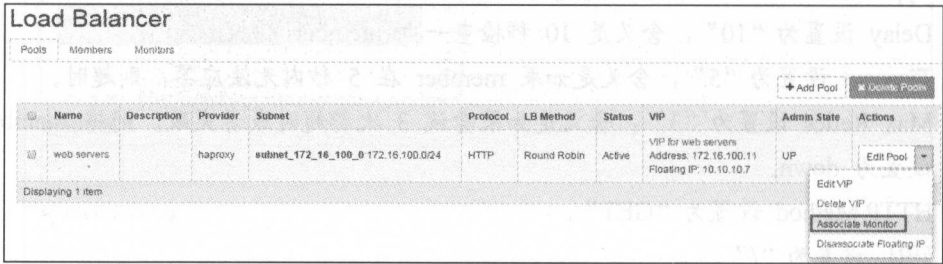


图 9-285

选择我们刚刚创建的 monitor, 如图 9-286 所示。

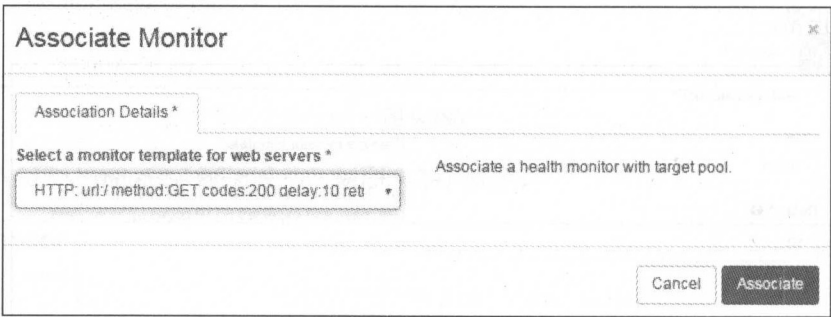


图 9-286

单击 “Associate” 完成配置。

## 8. 测试 LBaaS

经过上面的设置, 我们创建了包含 member “Web1” 和 “Web2” 的 Pool “web servers”, 并添加了 monitor。

准备就绪, 可以测试 Load Balancer 是否正常工作了。

首先在 Web1 和 Web2 中启动 HTTP 服务, 在 80 端口监听, 如图 9-287、图 9-288 所示。

```
ubuntu@web1:~$
ubuntu@web1:~$ echo "this is web1" > ~/index.html
ubuntu@web1:~$ sudo python -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
```

图-287

```
ubuntu@web2:~$
ubuntu@web2:~$ echo "This is web2" > ~/index.html
ubuntu@web2:~$ sudo python -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
```

图 9-288

这里我们使用 python 提供的 SimpleHTTPServer 模块启动了 HTTP 服务。

Web Server 的 index.html 显示当前访问的是哪个 member。

在 router 的 namespace 上多次执行 curl 172.16.100.11 (VIP)，如图 9-289 所示。

```
root@devstack-controller:~# ip netns exec qrouter-6246516b-
-be0d82e05f1a curl 172.16.100.11
This is web2
root@devstack-controller:~# ip netns exec qrouter-6246516b-
-be0d82e05f1a curl 172.16.100.11
This is web2
root@devstack-controller:~# ip netns exec qrouter-6246516b-
-be0d82e05f1a curl 172.16.100.11
```

图 9-289

测试结果显示每次访问的都是 Web2 这个 member。

为什么没有访问到 Web1 呢？

还记得我们前面讨论的内容吗：

Load Balance Method - ROUND\_ROUBIN

Session Persistence - SOURCE\_IP

在这种配置下，第一个 curl 请求 HAProxy 通过 ROUND\_ROUBIN 选择了 Web2。下面我们修改一下配置。而后续的请求，HAProxy 则会应用 SOURCE\_IP 机制，仍然选择 Web2。

在“Web Servers”的操作列表中单击“Edit VIP”，如图 9-290 所示。

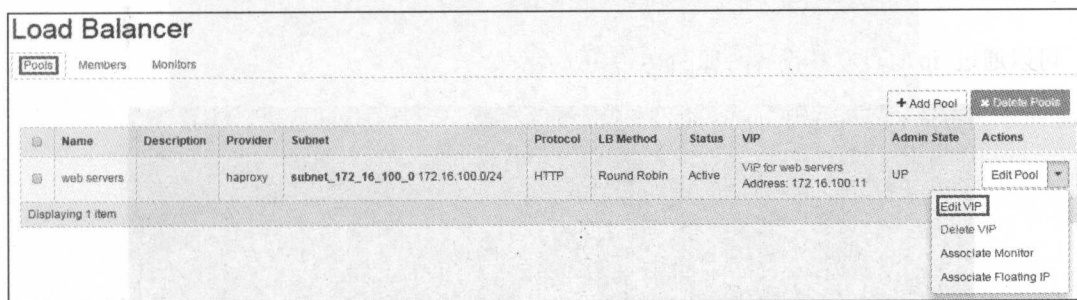


图 9-290

选择“No session persistence”并保存，如图 9-291 所示。

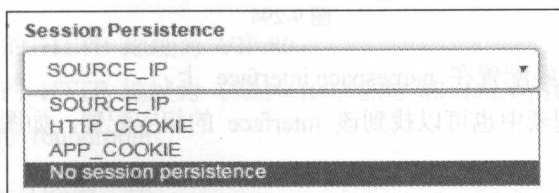


图 9-291

再进行 curl 测试，如图 9-292 所示。

```
root@devstack-controller:~# ip netns exec qrouter-6246516b-
-be0d82e05f1a curl 172.16.100.11
This is web2
root@devstack-controller:~# ip netns exec qrouter-6246516b-
-be0d82e05f1a curl 172.16.100.11
This is web1
root@devstack-controller:~# ip netns exec qrouter-6246516b-
-be0d82e05f1a curl 172.16.100.11
This is web2
root@devstack-controller:~# ip netns exec qrouter-6246516b-
-be0d82e05f1a curl 172.16.100.11
This is web1
```

图 9-292

可以看到已经在“Web1”和“Web2”之间 round robin 了。

## 9. 底层发生了什么变化

下面我们考察一下 Neutron 是如何实现 LBaaS 的。

在控制节点上运行 `ip netns`，我们发现 Neutron 创建了新的 namespace `qlbaas-xxx`。

该 namespace 对应我们创建的 pool “web servers”，其命名格式为 `qlbaas-<pool ID>`，如图 9-293 所示。

```
root@devstack-controller:~# ip netns
qdhcp-3e6f0acf-8fef-4de6-a889-3711f84ffcce
qlbaas-a872c1fe-4044-42f4-8ef8-76dad02ff635
qrouter-6246516b-96d7-4520-81d7-be0d82e05f1a
qdhcp-c7bb27c7-bdf2-4ed8-9f75-32cc93de945d
qdhcp-7e95400c-d094-4de9-bb6d-b5d25effe362
root@devstack-controller:~#
```

图 9-293

可以通过 `ip a` 查看其设置，如图 9-294 所示。

```
root@devstack-controller:~# ip netns exec qlbaas-a872c1fe-4044-42f4-8ef8-
76dad02ff635 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: ns-259ee763-8f: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo
_fast state UP group default qlen 1000
    link/ether fa:16:3e:24:bf:64 brd ff:ff:ff:ff:ff:ff
    inet 172.16.100.11/24 brd 172.16.100.255 scope global ns-259ee763-8f
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe24:bf64/64 scope link
        valid_lft forever preferred_lft forever
```

图 9-294

VIP 172.16.100.11 已经配置在 namespace interface 上。

在 subnet 的 Port 列表表中也可以找到该 interface 的相应配置，如图 9-295 所示。



Ports		
Name	Fixed IPs	Attached Device
(297924eb-61c8)	172.16.100.2	network:dhcp
vip-11256de2-6290-43e7-a3b0-8b2be3f834e9	172.16.100.11	neutron:LOADBALANCER
(e7355d33-a690)	172.16.100.10	compute:nova
(e23f8ecc-c020)	172.16.100.1	network:router_interface
(fa1dc7a7-26f5)	172.16.100.9	compute:nova
Displaying 5 items		

图 9-295

对于每一个 pool，Neutron 都会启动一个 haproxy 进程提供 load balancering 功能。通过 ps 命令查找 haproxy 进程，如图 9-296 所示。

```
root@devstack-controller:~# ps -elf|grep -v agent|grep haproxy
1 S nobody 28894 1 0 80 0 - 5094 ep_pol 19:30 ?
00:00:00 haproxy -f /opt/stack/data/neutron/lbaas/a872c1fe-404
4-42f4-8ef8-76dad02ff635/conf -p /opt/stack/data/neutron/lbaas/a
872c1fe-4044-42f4-8ef8-76dad02ff635/pid -sf 28829
```

图 9-296

haproxy 配置文件保存在 /opt/stack/data/neutron/lbaas/<pool ID>/conf 中。

查看 Web Servers 的配置文件内容，如图 9-297 所示。

```
global
    daemon
    user nobody
    group nogroup
    log /dev/log local0
    log /dev/log local1 notice
    stats socket /opt/stack/data/neutron/lbaas/a872c1fe-42f4-8ef8-76dad
02ff635/sock mode level:user
defaults
    log global
    retries
    option redispatch
    timeout connect
    timeout client
    timeout server
frontend 11256de2-a3b0-8b2be3f834e9
    option tcplog
    bind
    mode http
    default_backend a872c1fe-42f4-8ef8-76dad02ff635
    option forwardfor
backend a872c1fe-42f4-8ef8-76dad02ff635
    mode http
    balance roundrobin
    option forwardfor
    server b4ca5104-b958-44b3-a575-cf708b6f4b42 weight
    server 1e5f0925-3e6e-4c22-81eb-ab97681c0653 weight
```

图 9-297

可以看到：

- (1) frontend 使用的 HTTP 地址为 VIP:80。
- (2) backend 使用的 HTTP 地址为 172.16.100.10:80 和 172.16.100.9:80。
- (3) balance 方法为 roundrobin。

这些内容与我们前面的配置一致。

10. 通过 floating IP 访问 VIP

前面我们是直接用 curl 测试 VIP，在更为真实的场景中通常会使用 floating IP 访问 VIP。下面我们给 VIP 关联一个 floating IP，再进行测试。

访问 Project → Compute → Access & Security，打开 Floating IPs 标签页，单击“Allocate IP to Project”，如图 9-298 所示。

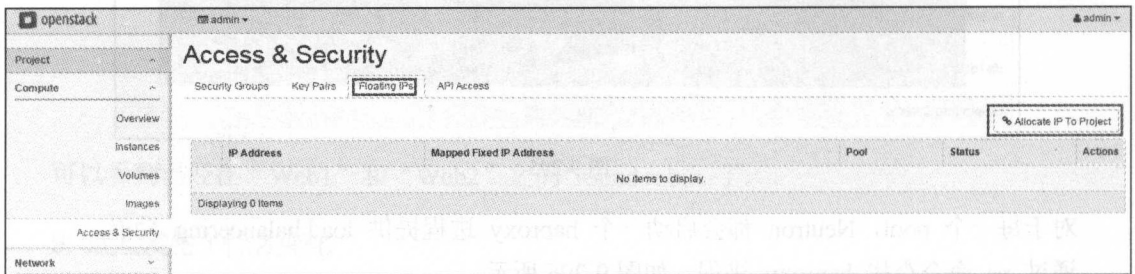


图 9-298

在 Pool 下拉列表中选择“ext\_net”，Neutron 将从该网络中分配 Floating IP，如图 9-299 所示。

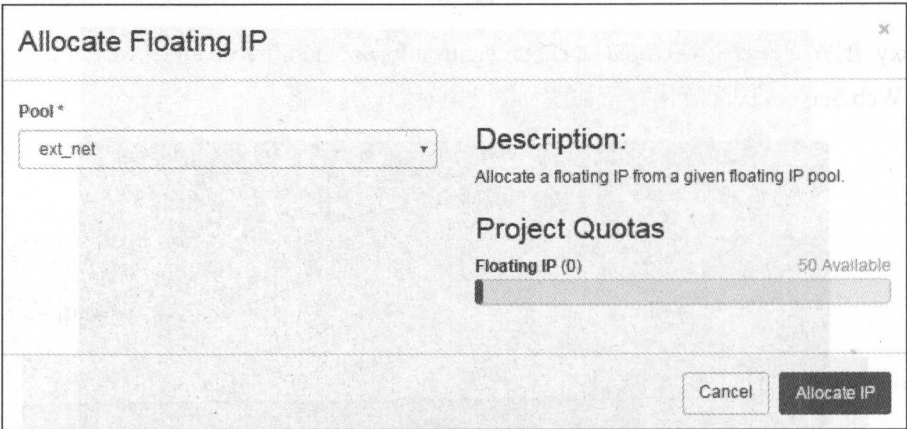


图 9-299

单击“Allocate IP”，如图 9-300 所示，分配到的 IP 为“10.10.10.7”。

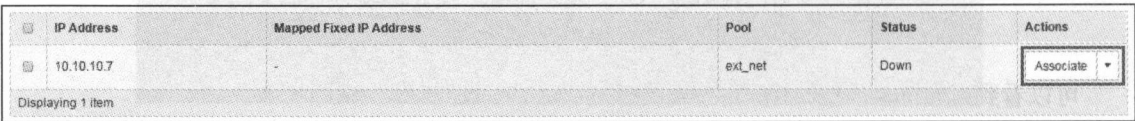


图 9-300

单击“Associate”，打开如图 9-301 所示的界面。

**Manage Floating IP Associations**

IP Address \*

IP Address \*  
10.10.10.7

Select the IP address you wish to associate with the selected instance or port.

Port to be associated \*

Select a port  
Select a port  
VIP for web servers: 172.16.100.11  
Web1: 172.16.100.9  
Web2: 172.16.100.10

Cancel Associate

图 9-301

在“Port to be associated”列表中选择“VIP for web servers: 172.16.100.11”，并单击“Associate”，打开如图 9-302 所示界面。

IP Address	Mapped Fixed IP Address	Pool	Status
10.10.10.7	Load Balancer VIP 172.16.100.11	ext_net	Active

Displaying 1 item

图 9-302

成功将外网 IP 10.10.10.7 关联到 VIP。

在 IP 为 10.10.10.4 的 instance 中进行 curl 测试，如图 9-303 所示。

```
$ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr EA:16:3E:4F:A4:E4
          inet addr:10.10.10.4  Bcast:10.10.10.255  Mask:255.255.255.0
          inet6 addr: fe80::f816:3eff:fe4f:a4e4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:41 errors:0 dropped:0 overruns:0 frame:0
          TX packets:41 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3440 (3.3 KiB)  TX bytes:3311 (3.2 KiB)

$ curl 10.10.10.7
This is Web2
$ curl 10.10.10.7
This is Web1
$ curl 10.10.10.7
This is Web2
$ curl 10.10.10.7
This is Web1
```

图 9-303

Floating IP 生效，load balancer 工作正常。

## 11. 小节

LBaaS 为租户提供了横向扩展应用的能力。

租户可以将外部请求 balancing 到多个 instance 上，并通过 monitor 实现应用的高可用。

LBaaS 当前的实现是基于 HAProxy，其功能已经能够满足普通业务需求。

# 9.5 Open vSwitch 实现 Neutron 网络

Linux Bridge 和 Open vSwitch 是目前 OpenStack 中使用最广泛的两种虚机交换机技术。

前面各章节我们已经学习了如何用 Linux Bridge 作为 ML2 mechanism driver 实现 Neutron 网络。本节我们将详细讨论如何用 Open vSwitch 实现 Neutron。

实验环境中两节点的网卡分配方式与 Linux Bridge 一致，如图 9-304 所示。

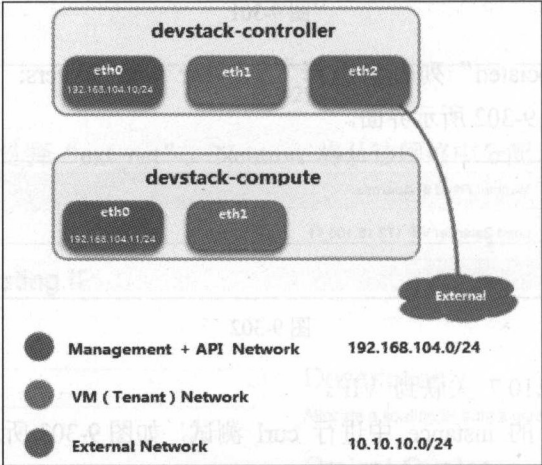


图 9-304

- 控制节点三个网卡 (eth0, eth1, eth2)，计算节点两网卡 (eth0, eth1)。
- 合并 Management 和 API 网络，使用 eth0，IP 段为 192.168.104.0/24。
- VM 网络使用 eth1。
- 控制节点的 eth2 与 External 网络连接，IP 段为 10.10.10.0/24。

## 9.5.1 网络拓扑

实验环境的网络拓扑如图 9-305 所示。



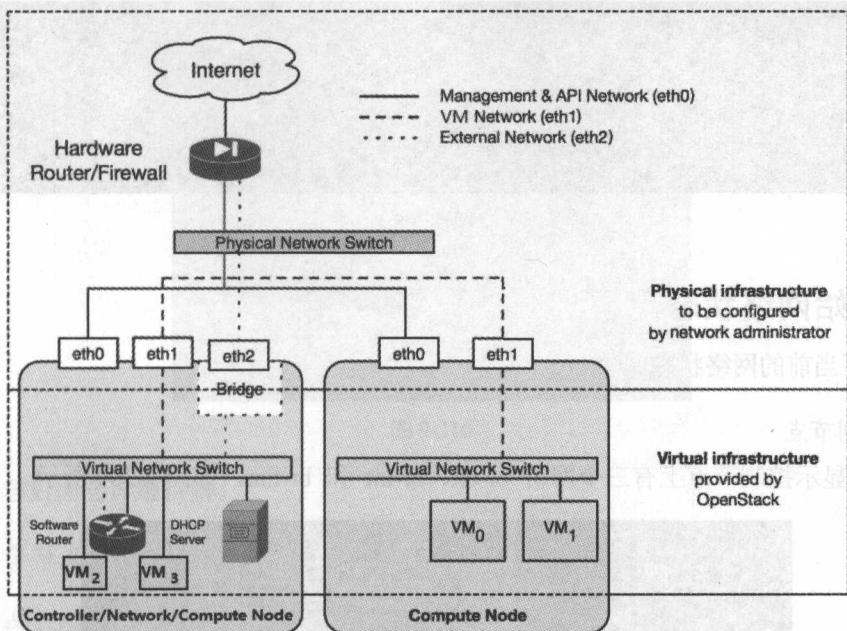


图 9-305

这个图在 Linux Bridge 实现中也看到过，唯一的区别是：对于节点中的“Virtual Network Switch”我们将用 Open vSwitch 替换掉 Linux Bridge。

## 9.5.2 配置 openvswitch mechanism driver

要将 Linux Bridge 切换到 Open vSwitch，首先需要安装 Open vSwitch 的 agent。修改 devstack 的 local.conf，如图 9-306 所示。

```
Q_AGENT=openvswitch
```

图 9-306

重新运行 ./stack，devstack 会自动下载并安装 Open vSwitch。

接下来就可以修改 ML2 的配置文件 /etc/neutron/plugins/ml2/ml2\_conf.ini，设置使用 openvswitch mechanism driver，如图 9-307 所示。

```
tenant_network_types = local
extension_drivers = port_security
type_drivers = local,flat,vlan,gre,vxlan
mechanism_drivers = openvswitch
```

图 9-307

控制节点和计算节点都需要按照上面的方法安装并配置 Open vSwitch。

Neutron 服务重启后，可以通过 neutron agent-list 命令查看到 neutron-openvswitch-agent 已经在两个节点上运行，如图 9-308 所示。

agent_type	host	alive	admin_state_up	binary
Metadata agent	devstack-controller	--	True	neutron-metadata-agent
Open vSwitch agent	devstack-compute1	--	True	neutron-openvswitch-agent
Loadbalancer agent	devstack-controller	--	True	neutron-lbaas-agent
DHCP agent	devstack-controller	--	True	neutron-dhcp-agent
Open vSwitch agent	devstack-controller	--	True	neutron-openvswitch-agent
L3 agent	devstack-controller	--	True	neutron-vpn-agent

图 9-308

9.5.3 初始网络状态

查看一下当前的网络状态。

- 控制节点

ifconfig 显示控制节点上有三个网桥 br-ex、br-int 和 br-tun，如图 9-309 所示。

```
root@devstack-controller:~# ifconfig
br-ex      Link encap:Ethernet  Hwaddr 9e:8c:1d:18:a0:4a
           inet6 addr: fe80::588b:fbff:fec3:3432/64 Scope:Link
           UP BROADCAST RUNNING MTU:1500 Metric:1
           RX packets:0 errors:0 dropped:0 overruns:0 frame:0
           TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:0
           RX bytes:0 (0.0 B)  TX bytes:648 (648.0 B)

br-int     Link encap:Ethernet  Hwaddr 26:37:f6:36:16:40
           inet6 addr: fe80::64ea:c7ff:fe30:29f8/64 Scope:Link
           UP BROADCAST RUNNING MTU:1500 Metric:1
           RX packets:0 errors:0 dropped:0 overruns:0 frame:0
           TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:0
           RX bytes:0 (0.0 B)  TX bytes:648 (648.0 B)

br-tun     Link encap:Ethernet  Hwaddr f6:1a:e2:dd:08:4d
           inet6 addr: fe80::cc7d:55ff:fed5:df44/64 Scope:Link
           UP BROADCAST RUNNING MTU:1500 Metric:1
           RX packets:0 errors:0 dropped:0 overruns:0 frame:0
           TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:0
           RX bytes:0 (0.0 B)  TX bytes:648 (648.0 B)
```

图 9-309

从命名上看我们大致能猜出他们的用途：

- br-ex

连接外部（external）网络的网桥。

- br-int

集成（integration）网桥，所有 instance 的虚拟网卡和其他虚拟网络设备都将连接到该网桥。

- br-tun

隧道（tunnel）网桥，基于隧道技术的 VxLAN 和 GRE 网络将使用该网桥进行通信。

这些网桥都是 Neutron 自动为我们创建的，但是通过 brctl show 命令却看不到它们。

这是因为我们使用的是 Open vSwitch 而非 Linux Bridge，需要用 Open vSwitch 的命令 ovs-vsctl show 查看，如图 9-310 所示。

```

root@devstack-controller:~# ovs-vsctl show
c3f35829-cb18-41e5-b2e1-d378aaa9351d
    Bridge br-ex
        Port br-ex
            Interface br-ex
                type: internal
    Bridge br-int
        fail_mode: secure
        Port br-int
            Interface br-int
                type: internal
    Bridge br-tun
        fail_mode: secure
        Port br-tun
            Interface br-tun
                type: internal
        Port patch-int
            Interface patch-int
                type: patch
            options: {peer=patch-tun}
    ovs_version: "2.0.2"
root@devstack-controller:~#

```

图 9-310

输出内容后面会详细讲解。

### ● 计算节点

计算节点上也有 br-int 和 br-tun，但没有 br-ext。

这是合理的，因为发送到外网的流量是通过网络节点上的虚拟路由器转发出去的，所以 br-ext 只会放在网络节点（devstack-controller）上，如图 9-311、图 9-312 所示。

```

root@devstack-compute1:~# ifconfig
br-int: Link encap:Ethernet Hwaddr a6:cf:b8:20:0a:43
        inet6 addr: fe80::50e1:ebff:fe13:b093/64 Scope:Link
        UP BROADCAST RUNNING MTU:1500 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:0 (0.0 B) TX bytes:648 (648.0 B)

br-tun: Link encap:Ethernet Hwaddr b6:14:f7:de:c0:4e
        inet6 addr: fe80::c84f:89ff:feb7:12/64 Scope:Link
        UP BROADCAST RUNNING MTU:1500 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:0 (0.0 B) TX bytes:648 (648.0 B)

```

图 9-311

```

root@devstack-compute1:~# ovs-vsctl show
c843c12c-9c70-4e26-99c9-ce58e3997fc6
    Bridge br-int
        fail_mode: secure
        Port br-int
            Interface br-int
                type: internal
    Bridge br-tun
        fail_mode: secure
        Port patch-int
            Interface patch-int
                type: patch
            options: {peer=patch-tun}
        Port br-tun
            Interface br-tun
                type: internal
    ovs_version: "2.0.2"
root@devstack-compute1:~#

```

图 9-312

## 9.5.4 了解 Open vSwitch 环境中的各种网络设备

在 Open vSwitch 环境中，一个数据包从 instance 发送到物理网卡大致会经过下面几个类型的设备：

- (1) tap interface，命名为 tapXXXX。
- (2) linux bridge，命名为 qbrXXXX。
- (3) veth pair，命名为 qvbXXXX, qvoXXXX。
- (4) OVS integration bridge，命名为 br-int。
- (5) OVS patch ports，命名为 int-br-ethX 和 phy-br-ethX (X 为 interface 的序号)。
- (6) OVS provider bridge，命名为 br-ethX (X 为 interface 的序号)。
- (7) 物理 interface，命名为 ethX (X 为 interface 的序号)。
- (8) OVS tunnel bridge，命名为 br-tun。

OVS provider bridge 会在 flat 和 vlan 网络中使用；OVS tunnel bridge 则会在 vxlan 和 gre 网络中使用。

后面会通过实例详细讨论这些设备。

Open vSwitch 支持 local、flat、vlan、vxlan 和 gre 五种 network type。

vxlan 和 gre 非常类似，接下来我们将深入学习 Open vSwitch 是如何实现 local、flat、vlan 和 vxlan 的。

## 9.5.5 local network

local network 不会与宿主机的任何物理网卡连接，流量只被限制在宿主机内，同时也不关联任何的 VLAN ID。

### 1. 创建第一个 local network

下面我们通过 Web GUI 创建 local network。

进入菜单 Admin → Networks，单击“Create Network”按钮，如图 9-313 所示。

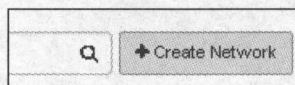
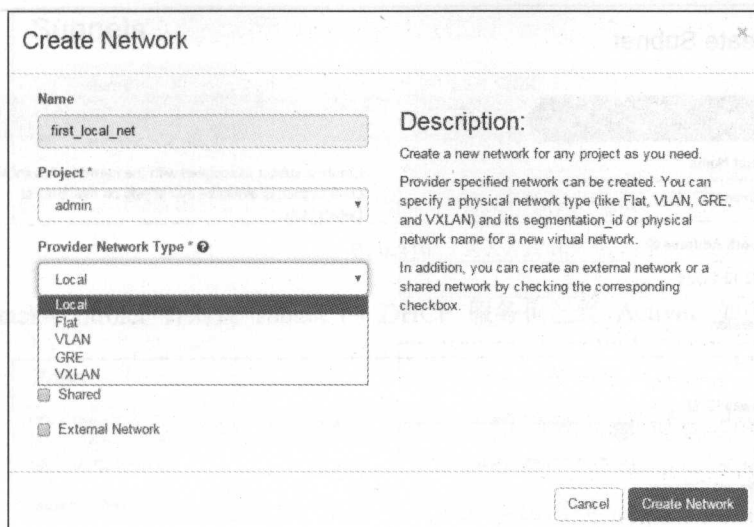


图 9-313

显示创建页面，如图 9-314 所示。





**Create Network**

Name: first\_local\_net

Project: admin

Provider Network Type: Local (selected from dropdown: Local, Flat, VLAN, GRE, VXLAN)

☐ Shared

☐ External Network

**Description:**  
 Create a new network for any project as you need.  
 Provider specified network can be created. You can specify a physical network type (like Flat, VLAN, GRE, and VXLAN) and its segmentation\_id or physical network name for a new virtual network.  
 In addition, you can create an external network or a shared network by checking the corresponding checkbox.

Cancel Create Network

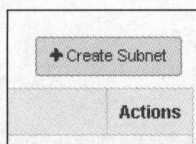
图 9-314

在“Provider Network Type”下拉列表框中选择“Local”，单击“Create Network”，first\_local\_net 创建成功，如图 9-315 所示。

<input type="checkbox"/>	Project	Network Name
<input type="checkbox"/>	admin	first_local_net
Displaying 1 item		

图 9-315

单击 first\_local\_net 链接，进入 network 配置页面，目前还没有 subnet，单击“Create Subnet”按钮，如图 9-316 所示。



+ Create Subnet

Actions

图 9-316

设置 IP 地址为“172.16.1.0/24”，如图 9-317 所示。

The screenshot shows the 'Create Subnet' form in OpenStack. The first step, 'Subnet', is highlighted in the progress bar. The form contains the following fields and options:

- Subnet Name:** A text input field containing 'subnet\_172\_16\_1\_0'.
- Network Address:** A text input field containing '172.16.1.0/24'.
- IP Version:** A dropdown menu set to 'IPv4'.
- Gateway IP:** An empty text input field.
- Disable Gateway:** A checkbox that is currently unchecked.

On the right side of the form, there is a note: 'Create a subnet associated with the network. Advanced configuration is available by clicking on the "Subnet Details" tab.' At the bottom, there are two buttons: 'Back' and 'Next'.

图 9-317

单击“Next”，如图 9-318 所示。

The screenshot shows the 'Create Subnet' form in OpenStack, now at the 'Subnet Details' step. The progress bar has moved to the second step. The form contains the following fields and options:

- Enable DHCP:** A checkbox that is checked.
- Allocation Pools:** A text input field containing '172.16.1.2,172.16.1.99'.
- DNS Name Servers:** An empty text input field.
- Host Routes:** An empty text input field.

On the right side of the form, there is a note: 'Specify additional attributes for the subnet.' At the bottom, there are two buttons: 'Back' and 'Create'.

图 9-318



选中“Enable DHCP”，IP 池设置为“172.16.1.2,172.16.1.99”。单击“Create”，subnet 创建成功，如图 9-319 所示。

Subnets		
<input type="checkbox"/>	Name	CIDR
<input type="checkbox"/>	subnet_172_16_1_0	172.16.1.0/24
Displaying 1 item		

图 9-319



同时 devstack-controller 针对此 subnet 的 DHCP 服务也已经 Active，如图 9-320 所示。

Ports

	Name	Fixed IPs	Attached Device	Status
	(7970bdcd-f2cd)	172.16.1.2	network:dhcp	Active

Displaying 1 item

DHCP Agents

	Host	Status	Admin State
	devstack-controller	Enabled	Up

Displaying 1 item

图 9-320

底层网络发生了什么变化

network “local\_net” 已经创建成功了，下面我们需要搞清楚底层网络结构有了哪些变化？

打开控制节点的 shell 终端，用 `ovs-vsctl show` 查看当前 Open vSwitch 的状态，如图 9-321 所示。

```

root@devstack-controller:~#
root@devstack-controller:~# ovs-vsctl show
c3f35829-cb18-41e5-b2e1-d378aaa9351d
    Bridge br-ex
        Port br-ex
            Interface br-ex
                type: internal
    Bridge br-int
        fail_mode: secure
        Port br-int
            Interface br-int
                type: internal
        Port "tap7970bdcd-f2"
            tag: 1
            Interface "tap7970bdcd-f2"
                type: internal
    Bridge br-tun
        fail_mode: secure
        Port br-tun
            Interface br-tun
                type: internal
        Port patch-int
            Interface patch-int
                type: patch
            options: {peer=patch-tun}
    ovs_version: "2.0.2"
root@devstack-controller:~#

```

图 9-321

可以看到 Neutron 自动在 br-int 网桥上创建了 port “tap7970bdcd-f2”。

从命名可知，该 port 对应 local\_net 的 dhcp 接口。

与 linux bridge driver 一样，dhcp 设备也是放在命名空间里的，如图 9-322 所示。

```
root@devstack-controller:~# ip netns
qdhcp-07c8b9dd-96cb-484c-ad85-c52ac655c6ff
root@devstack-controller:~#
root@devstack-controller:~# ip netns exec qdhcp-07c8b9dd-96cb-484c-ad85-c52ac655c6ff ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
10: tap7970bdc-d-f2: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default
    link/ether fa:16:3e:ff:e8:e3 brd ff:ff:ff:ff:ff:ff
    inet 172.16.1.2/24 brd 172.16.1.255 scope global tap7970bdc-d-f2
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:feff:e8e3/64 scope link
        valid_lft forever preferred_lft forever
root@devstack-controller:~#
```

图 9-322

目前网络结构如图 9-323 所示。

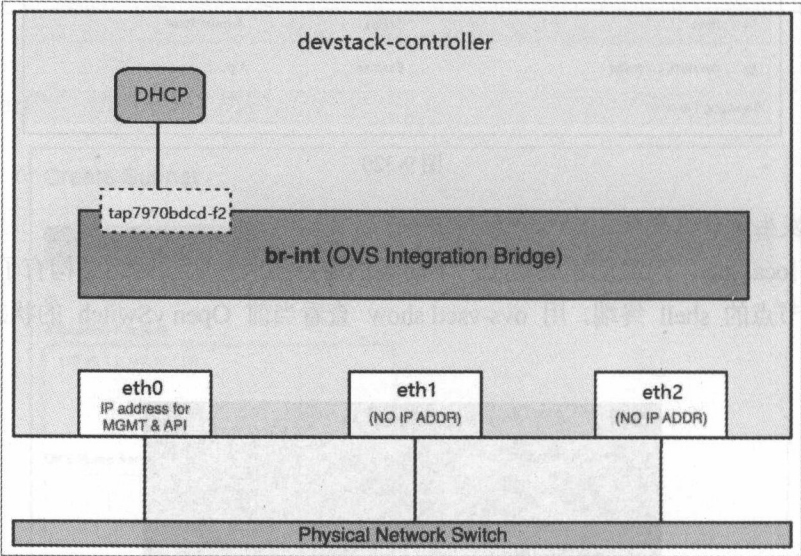


图 9-323

2. 将 instance 连接到 first\_local\_net

Launch 一个 Instance，选择 first\_local\_net 网络，如图 9-324 所示。



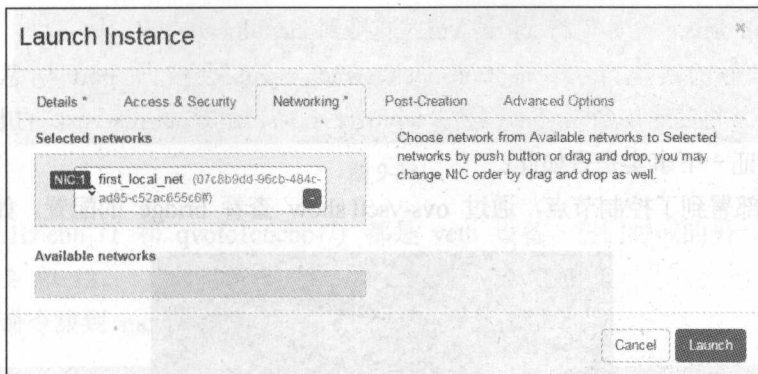


图 9-324

instance 部署成功，分配的 IP 地址为 172.16.1.3，如图 9-325 所示。

<input type="checkbox"/>	Instance Name	Image Name	IP Address
<input type="checkbox"/>	cirros-vm1	cirros	172.16.1.3
Displaying 1 item			

图 9-325

底层网络发生了什么变化

对于 instance “cirros-vm1”，Neutron 会在 subnet 中创建一个 port，分配 IP 和 MAC 地址，并将 port 分配给 cirros-vm1，如图 9-326 所示。

Ports		
Name	Fixed IPs	Attached Device
(fc1c6ebb-719d)	172.16.1.3	compute:nova
(7970bdc-d2cd)	172.16.1.2	network:dhcp
Displaying 2 items		

图 9-326

如图 9-326 所示，port 列表中增加了一个 port “(fc1c6ebb-719d)”，IP 为 172.16.1.3，单击 port 名称查看 MAC 信息，如图 9-327 所示。

Port Details	
Name	None
ID	fc1c6ebb-719d-412d-81d6-b2bf299356f5
Network ID	07c8b9dd-96cb-484c-ad85-c52ac655c6ff
Project ID	2c23c5fedd334ee4903716f471ba405d
MAC Address	fa:16:3e:99:11:ac
Status	Active
Admin State	UP

图 9-327

我们可以先按照在 Linux Bridge Driver 章节学到的知识推测一下：

Open vSwitch driver 会如何将 cirros-vm1 连接到 first\_local\_net?

如果采用类似的实现方法, neutron-openvswitch-agent 会根据 port 信息创建 tap 设备 tapfc1c6ebb-71, 并将其连接到 br-int 网桥, tapfc1c6ebb-71 就是 cirros-vm1 的虚拟网卡。

下面我们验证一下事实是否如此:

cirros-vm1 部署到了控制节点, 通过 ovs-vsctl show 查看 bridge 的配置, 如图 9-328 所示。

```
root@devstack-controller:~# ovs-vsctl show
c3f35829-cb18-41e5-b2e1-d378aaa9351d
Bridge br-ex
  Port br-ex
    Interface br-ex
      type: internal
Bridge br-int
  fail_mode: secure
  Port br-int
    Interface br-int
      type: internal
  Port "tap7970bdcd-f2"
    tag: 1
    Interface "tap7970bdcd-f2"
      type: internal
  Port "qvofc1c6ebb-71"
    tag: 1
    Interface "qvofc1c6ebb-71"
```

图 9-328

非常遗憾, 在 br-int 上并没有看到 tapfc1c6ebb-71, 而是多了一个 qvofc1c6ebb-71。

目前我们并不知道 qvofc1c6ebb-71 是什么, 我们再用 brctl show 查看一下 Linux Bridge 的配置, 如图 9-329 所示。

```
root@devstack-controller:~# brctl show
bridge name      bridge id        STP enabled      interfaces
qbrfc1c6ebb-71   8000.42b062b884e6  no                qvbf1c6ebb-71
                  tapfc1c6ebb-71
virbr0           8000.000000000000  yes
```

图 9-329

这里我们看到有一个新建的网桥 qbrfc1c6ebb-71, 上面连接了两个设备 qvbf1c6ebb-71 和 tapfc1c6ebb-71。从命名上看, 他们都应该与 cirros-vm1 的虚拟网卡有关。

通过 virsh edit 查看 cirros-vm1 的配置, 如图 9-330 所示。

```
<interface type= bridge >
  <mac address= fa:16:3e:9b:31:ac />
  <source bridge= qbrfc1c6ebb-71 />
  <target dev= tapfc1c6ebb-71 />
  <model type= virtio />
```

图 9-330

确实 tapfc1c6ebb-71 是 cirros-vm1 的虚拟网卡。

那么 linux bridge qbrfc1c6ebb-71 上的 qvbf1c6ebb-71 设备与 Open vSwitch br-int 上的 qvofc1c6ebb-71 是什么关系呢?

下面的内容稍微需要一些技巧了。

我们用 ethtool -S 分别查看 qvbf1c6ebb-71 和 qvofc1c6ebb-71 的 statistics, 如图 9-331 所示。

```

root@devstack-controller:~# ethtool -S qvbf1c6ebb-71
NIC statistics:
  peer_ifindex: 12
root@devstack-controller:~# ethtool -S qvof1c6ebb-71
NIC statistics:
  peer_ifindex: 13

```

图 9-331

原来 qvbf1c6ebb-71 和 qvof1c6ebb-71 都是 veth 设备，它们对应的另一端 veth 设备的 index 分别是 12 和 13。

通过 ip a 命令找到 index 12 和 13 的设备，如图 9-332 所示。

```

12: qvof1c6ebb-71: <BROADCAST,MULTICAST,PROMISC,UP,LOWER_UP>
    link/ether c2:ad:20:87:ef:e9 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::c0ad:20ff:fe87:efe9/64 scope link
        valid_lft forever preferred_lft forever
13: qvbf1c6ebb-71: <BROADCAST,MULTICAST,PROMISC,UP,LOWER_UP>
    link/ether 42:b0:62:b8:84:e6 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::40b0:62ff:feb8:84e6/64 scope link
        valid_lft forever preferred_lft forever

```

图 9-332

到这里，相信很多同学已经看出来了，qvbf1c6ebb-71 和 qvof1c6ebb-71 组成了一个 veth pair。我们之前介绍过，veth pair 是一种成对出现的特殊网络设备，它们像一根虚拟的网线连接两个网络设备。这里 qvbf1c6ebb-71 和 qvof1c6ebb-71 的作用就是连接网桥 qbrfc1c6ebb-71 和 br-int。

文字描述往往是不够直观的，下面我们将前面梳理好的信息通过图片展示出来，如图 9-333 所示。

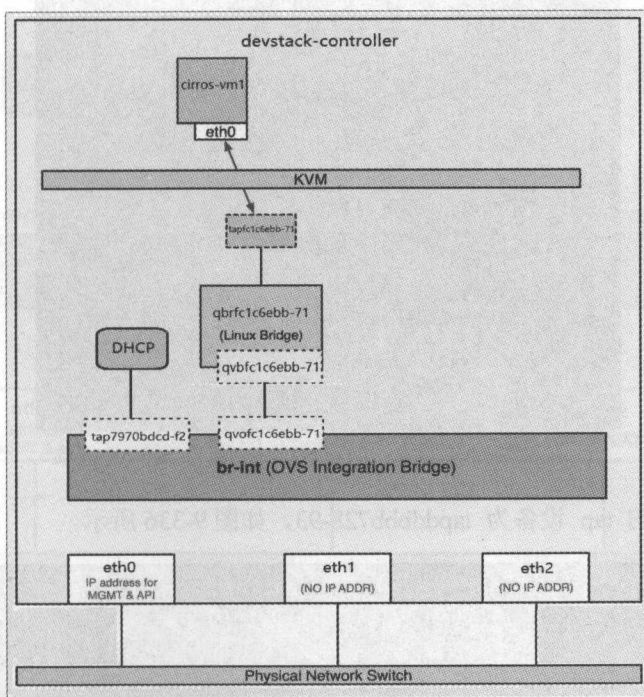


图 9-333

由图 9-333 所示，tapfc1c6ebb-71 通过 qbrfc1c6ebb-71 间接连接到 br-int。

那问题来了，为什么 tapfc1c6ebb-71 不能像左边的 DHCP 设备 tap7970bdcd-f2 那样直接连接到 br-int 呢？

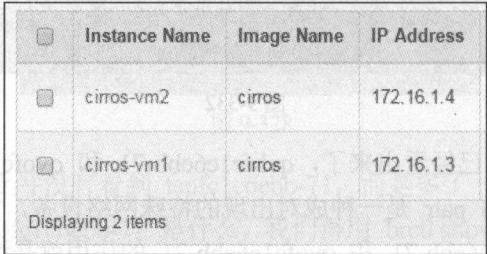
其原因是：Open vSwitch 目前还不支持将 iptables 规则放在与它直接相连的 tap 设备上。如果做不到这一点，就无法实现 Security Group 功能。

为了支持 Security Group，不得不多引入一个 Linux Bridge 支持 iptables。

这样的后果就是网络结构更复杂了，路径上多了一个 linux bridge 和 一对 veth pair 设备。

### 3. 连接第二个 instance 到 first\_local\_net

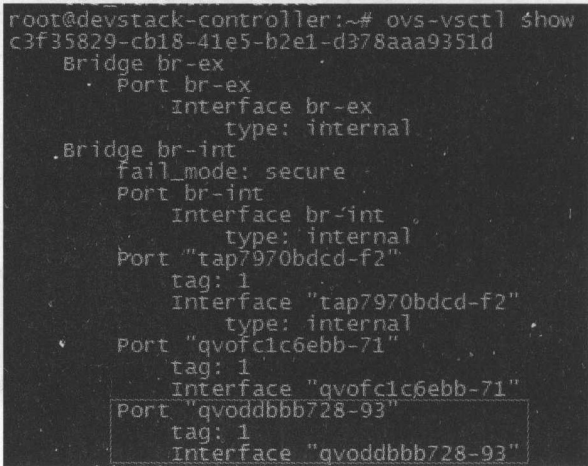
以同样的方式 launch instance “cirros-vm2”，分配的 IP 为 172.16.1.4，如图 9-334 所示。



<input type="checkbox"/>	Instance Name	Image Name	IP Address
<input type="checkbox"/>	cirros-vm2	cirros	172.16.1.4
<input type="checkbox"/>	cirros-vm1	cirros	172.16.1.3
Displaying 2 items			

图 9-334

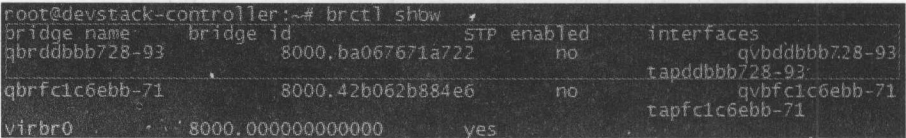
cirros-vm2 也被 schedule 到控制节点，ovs-vsctl show 的输出如图 9-335 所示。



```
root@devstack-controller:~# ovs-vsctl show
c3f35829-cb18-41e5-b2e1-d378aaa9351d
Bridge br-ex
  Port br-ex
    Interface br-ex
      type: internal
Bridge br-int
  fail_mode: secure
  Port br-int
    Interface br-int
      type: internal
  Port "tap7970bdcd-f2"
    tag: 1
    Interface "tap7970bdcd-f2"
      type: internal
  Port "qvofc1c6ebb-71"
    tag: 1
    Interface "qvofc1c6ebb-71"
  Port "qvoddbbb728-93"
    tag: 1
    Interface "qvoddbbb728-93"
```

图 9-335

cirros-vm2 对应的 tap 设备为 tapddbbb728-93，如图 9-336 所示。



bridge name	bridge id	STP enabled	interfaces
qbrddbbb728-93	8000.ba067671a722	no	qvoddbbb728-93
qbrfc1c6ebb-71	8000.42b062b884e6	no	qvofc1c6ebb-71
virbr0	8000.000000000000	yes	tapfc1c6ebb-71

图 9-336



从 cirros-vm2 能够 Ping 通 cirros-vm1 的 IP 地址 172.16.1.3, 如图 9-337 所示。

```
$ hostname
cirros-vm2
$ ping 172.16.1.3
PING 172.16.1.3 (172.16.1.3): 56 data bytes
64 bytes from 172.16.1.3: seq=0 ttl=64 time=2.352 ms
64 bytes from 172.16.1.3: seq=1 ttl=64 time=1.220 ms
64 bytes from 172.16.1.3: seq=2 ttl=64 time=1.341 ms
64 bytes from 172.16.1.3: seq=3 ttl=64 time=1.249 ms
64 bytes from 172.16.1.3: seq=4 ttl=64 time=2.382 ms
```

图 9-337

当前宿主机的网络结构, 如图 9-338 所示。

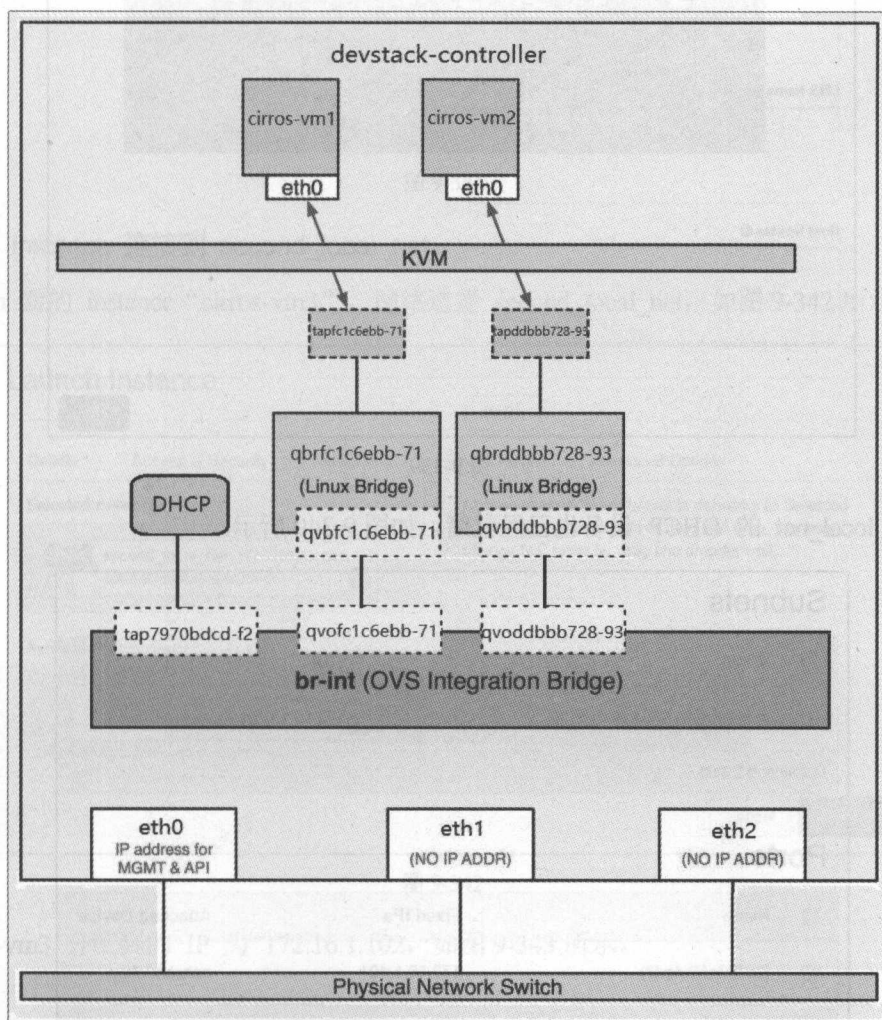


图 9-338

两个 instance 都挂在 br-int 上, 可以相互通信。

4. 创建第二个 local network

为了分析 local network 的连通性，我们再创建一个 “second\_local\_net”。second\_local\_net 的绝大部分属性与 first\_local\_net 相同，除了 IP 池范围为 172.16.1.101-172.16.1.200，如图 9-339 所示。

Create Subnet

Subnet Details

☒ Enable DHCP

Specify additional attributes for the subnet.

Allocation Pools

172.16.1.101,172.16.1.200

DNS Name Servers

Host Routes

Back

Create

图 9-339

second\_local\_net 的 DHCP 设备也已经就绪，如图 9-340 所示。

Subnets

<input type="checkbox"/>	Name	CIDR
<input type="checkbox"/>	subnet_172_16_1_0	172.16.1.0/24

Displaying 1 item

Ports

<input type="checkbox"/>	Name	Fixed IPs	Attached Device
<input type="checkbox"/>	(2c1b3c58-4ed3)	172.16.1.101	network:dhcp

Displaying 1 item

图 9-340

DHCP 对应的 tap 设备为 tap2c1b3c58-4e，已经连接到 br-int，如图 9-341 所示。

```

root@devstack-controller:~# ovs-vsctl show
c3f35829-cb18-41e5-b2e1-d378aaa9351d
    Bridge br-ex
        Port br-ex
            Interface br-ex
                type: internal
    Bridge br-int
        fail_mode: secure
        Port br-int
            Interface br-int
                type: internal
        Port "tap7970bdcd-f2"
            tag: 1
            Interface "tap7970bdcd-f2"
                type: internal
        Port "qvofc1c6ebb-71"
            tag: 1
            Interface "qvofc1c6ebb-71"
                type: internal
        Port "qvoddbbb728-93"
            tag: 1
            Interface "qvoddbbb728-93"
                type: internal
        Port "tap2c1b3c58-4e"
            tag: 2
            Interface "tap2c1b3c58-4e"
                type: internal

```

图 9-341

##### 5. 将 instance 连接到 second\_local\_net

launch 新的 instance “cirros-vm3”，网络选择 second\_local\_net，如图 9-342 所示。

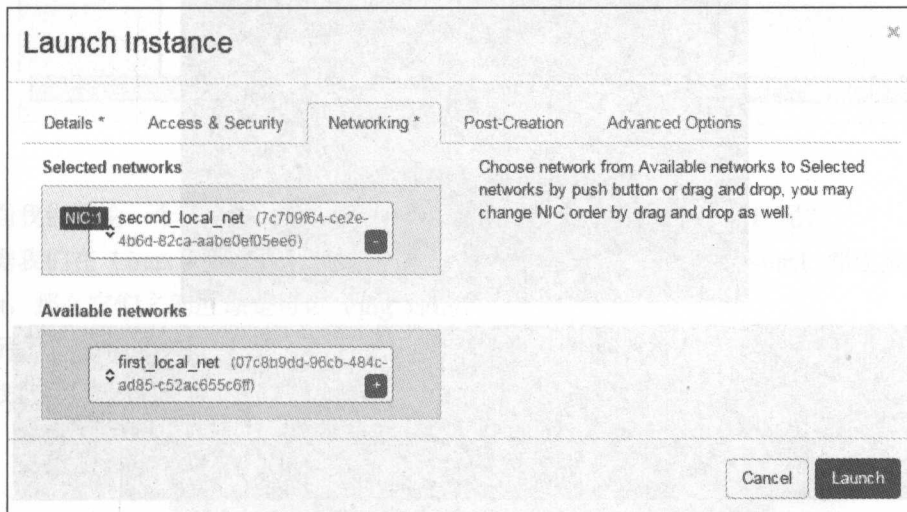


图 9-342

cirros-vm3 分配到的 IP 为 172.16.1.102，如图 9-343 所示。

<input type="checkbox"/>	Instance Name	Image Name	IP Address
<input type="checkbox"/>	cirros-vm3	cirros	172.16.1.102
<input type="checkbox"/>	cirros-vm2	cirros	172.16.1.4
<input type="checkbox"/>	cirros-vm1	cirros	172.16.1.3
Displaying 3 items			

图 9-343

cirros-vm3 被 schedule 到控制节点，其虚拟网卡也连接到 br-int，如图 9-344 所示。

```
root@devstack-controller:~# ovs-vsctl show
c3f35829-cb18-41e5-b2e1-d378aaa9351d
    Bridge br-ex
        Port br-ex
            Interface br-ex
                type: internal
    Bridge br-int
        fail_mode: secure
        Port br-int
            Interface br-int
                type: internal
        Port "qvofd5d4e5c-7e"
            tag: 2
            Interface "qvofd5d4e5c-7e"
        Port "tap7970bdc-d-f2"
            tag: 1
            Interface "tap7970bdc-d-f2"
                type: internal
        Port "qvofc1c6ebb-71"
            tag: 1
            Interface "qvofc1c6ebb-71"
        Port "qvoddbbb728-93"
            tag: 1
            Interface "qvoddbbb728-93"
        Port "tap2c1b3c58-4e"
            tag: 2
            Interface "tap2c1b3c58-4e"
                type: internal

root@devstack-controller:~# brctl show
bridge name      bridge id        STP enabled      interfaces
qbrddbbb728-93   8000.ba067671a722  no               qvbdbbb728-93
qbrfc1c6ebb-71   8000.42b062b884e6  no               tapddbbb728-93
qbrfd5d4e5c-7e   8000.da0b1f956388  no               qvbf1c6ebb-71
virbr0           8000.000000000000  yes              tapfd5d4e5c-7e
```

图 9-344

当前的控制节点上的网络结构如图 9-345 所示。



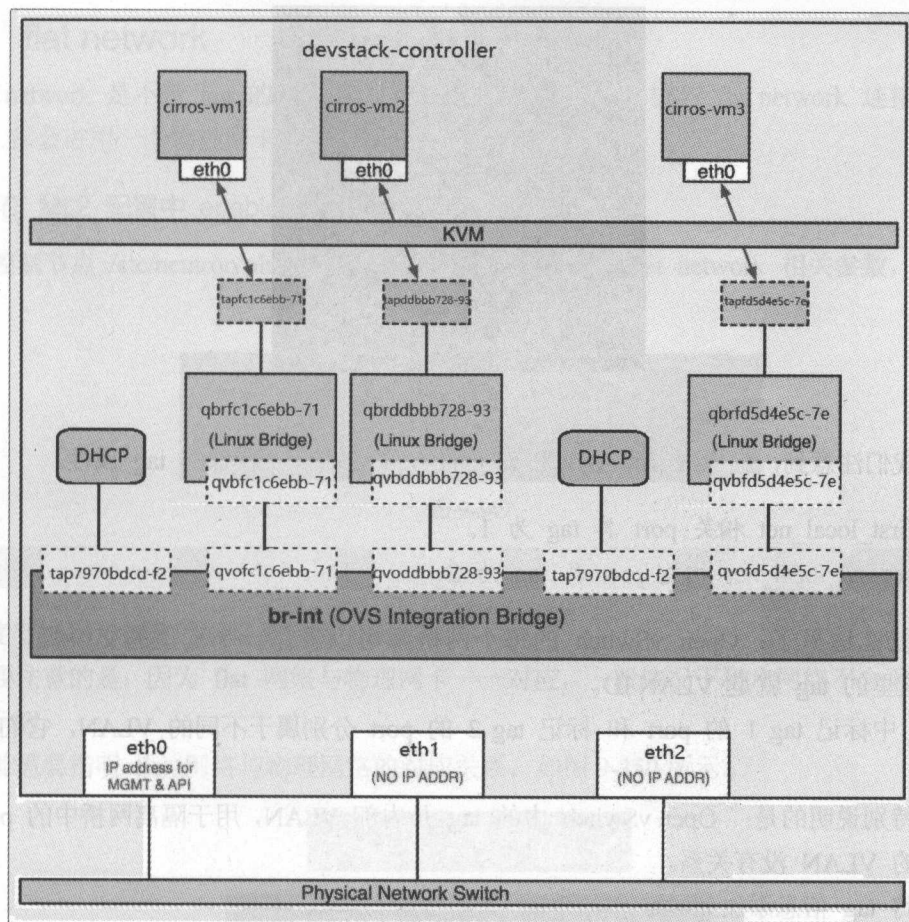


图 9-345

下面我们讨论一个有趣的问题：cirros-vm3 能否 Ping 到 cirros-vm1 呢？

根据我们在 Linux Bridge 中学到的知识，既然 cirros-vm3 和 cirros-vm1 都连接到同一个网桥 br-int，那么它们之间应该是可以 Ping 通的。

但另一方面，根据 Neutron 的设计，不同 local 网络之间是无法通信的。那么事实到底是如何呢？我们 ping 一下看看，结果如图 9-346 所示。

```
$ hostname
cirros-vm3
$ ping 172.16.1.3
PING 172.16.1.3 (172.16.1.3): 56 data bytes

--- 172.16.1.3 ping statistics ---
0 packets transmitted, 0 packets received, 100% packet loss
```

图 9-346

实验证明 cirros-vm3 无法 Ping 到 cirros-vm1。

下面我们需要解释同一个网桥上的 port 为什么不能通信。

让我们重新审视一下 br-int 上各个 port 的配置，如图 9-347 所示。

```
Bridge br-int
  fail_mode: secure
  Port br-int
    Interface br-int
      type: internal
    Port "qvofd5d4e5c-7e"
      tag: 2
      Interface "qvofd5d4e5c-7e"
    Port "tap7970bdcd-f2"
      tag: 1
      Interface "tap7970bdcd-f2"
      type: internal
    Port "qvofc1c6ebb-71"
      tag: 1
      Interface "qvofc1c6ebb-71"
    Port "qvoddbbb728-93"
      tag: 1
      Interface "qvoddbbb728-93"
    Port "tap2c1b3c58-4e"
      tag: 2
      Interface "tap2c1b3c58-4e"
      type: internal
```

图 9-347

这次我们注意到，虚拟网卡和 DHCP 对应的 port 都有一个特殊的 tag 属性。

- first\_local\_net 相关 port 其 tag 为 1。
- second\_local\_net 相关 port 其 tag 为 2。

玄机就在这里了：Open vSwitch 的每个网桥都可以看作一个真正的交换机，可以支持 VLAN，这里的 tag 就是 VLAN ID。

br-int 中标记 tag 1 的 port 和 标记 tag 2 的 port 分别属于不同的 VLAN，它们之间是隔离的。

需要特别说明的是：Open vSwitch 中的 tag 是内部 VLAN，用于隔离网桥中的 port，与物理网络中的 VLAN 没有关系。

我们将 tag 信息添加到网络结构图中，如图 9-348 所示。

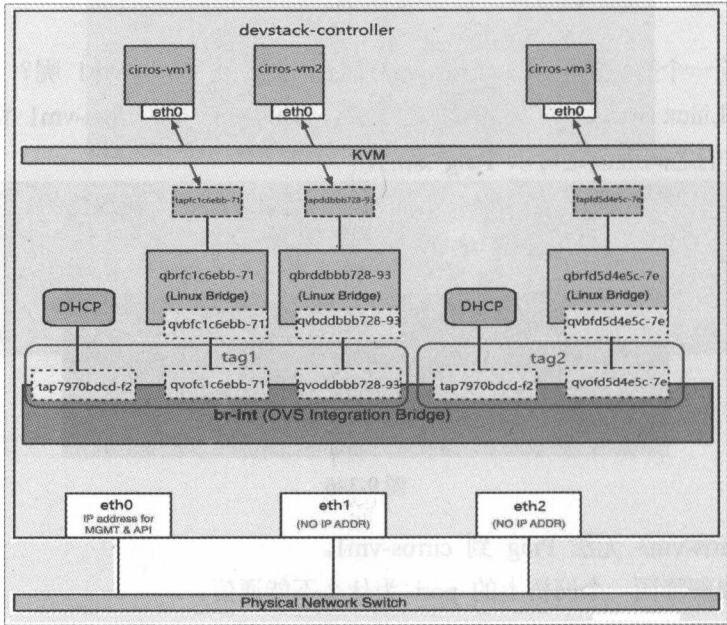


图 9-348

## 9.5.6 flat network

flat network 是不带 tag 的网络，宿主机的物理网卡通过网桥与 flat network 连接，每个 flat network 都会占用一个物理网卡。

### 1. 在 ML2 配置中 enable flat network

在控制节点 /etc/neutron/plugins/ml2/ml2\_conf.ini 中设置 flat network 相关参数，如图 9-349 所示。

```
[ml2]
tenant_network_types = flat
extension_drivers = port_security
type_drivers = local,flat,vlan,gre,vxlan
mechanism_drivers = openvswitch
```

图 9-349

```
tenant_network_types = flat
```

指定普通用户创建的网络类型为 flat。

需要注意的是：因为 flat 网络与物理网卡一一对应，一般情况下租户网络不会采用 flat，这里只是示例。

接着需要指明 flat 网络与物理网络的对应关系，如图 9-350 所示。

```
[ml2_type_flat]
flat_networks = default
```

```
[ovs]
bridge_mappings = default:br-eth1
```

图 9-350

如图 9-350 所示：

在 [ml2\_type\_flat] 中通过 flat\_networks 定义了一个 flat 网络，label 为 “default”。

在 [ovs] 中通过 bridge\_mappings 指明 default 对应的 Open vSwitch 网桥为 br-eth1。

label 是 flat 网络的标识，在创建 flat 时会用到（后面演示），label 的名字可以是任意字符串，只要确保各个节点 ml2\_conf.ini 中的 label 命名一致就可以了。

各个节点中 label 与物理网卡的对应关系可能不一样。这是因为每个节点可以使用不同的物理网卡将 instance 连接到 flat network。

与 linux bridge 实现的 flat 网络不同，ml2 中并不会直接指定 label 与物理网卡的对应关系，而是指定 label 与 ovs bridge 的对应关系。

```
[ovs]
bridge_mappings = default:br-eth1
```

这里的 ovs bridge 是 br-eth1，我们需要提前通过 ovs-ovctl 命令：

- 创建 br-eth1。
- 将物理网卡 eth1 桥接在 br-eth1 上，如图 9-351 所示。

```
root@devstack-controller:~# ovs-vsctl add-br br-eth1
root@devstack-controller:~#
root@devstack-controller:~# ovs-vsctl add-port br-eth1 eth1
root@devstack-controller:~#
```

图 9-351

如果要创建多个 flat 网络，需要定义多个 label，用逗号隔开，当然也需要用到多个 ovs bridge，如下所示：

```
[ml2_type_flat]
flat_networks = flat1,flat2
[ovs]
bridge_mappings = flat1:br-eth1,flat2:br-eth2
```

通过以上步骤控制节点的 flat 网络就准备好了。

计算节点也需要做相同的配置，然后重启所有节点的 Neutron 服务。

下面有必要通过 ovs-vsctl show 检视一下当前的网络结构，如图 9-352 所示。

```
root@devstack-controller:~# ovs-vsctl show
c3f35829-cb18-41e5-b2e1-d378aaa9351d
    Bridge br-ex
        Port br-ex
            Interface br-ex
                type: internal
    Bridge br-int
        fail_mode: secure
        Port br-int
            Interface br-int
                type: internal
        Port "int-br-eth1"
            Interface "int-br-eth1"
                type: patch
                options: {peer="phy-br-eth1"}
    Bridge "br-eth1"
        Port "phy-br-eth1"
            Interface "phy-br-eth1"
                type: patch
                options: {peer="int-br-eth1"}
        Port "eth1"
            Interface "eth1"
        Port "br-eth1"
            Interface "br-eth1"
                type: internal
```

图 9-352

对于 ovs bridge “br-eth1” 和其上桥接的 port “eth1” 我们应该不会感到意外，这是前面配置的结果。然而除此之外，br-int 和 br-eth1 分别多了一个 port “int-br-eth1” 和 “phy-br-eth1”，而且这两个 port 都是 “patch” 类型，同时通过 “peer” 指向对方。

上面的配置描述了这样一个事实：br-int 与 br-eth1 这两个网桥通过 int-br-eth1 和 phy-br-eth1 连接在一起了。

目前控制节点网络结构如图 9-353 所示。



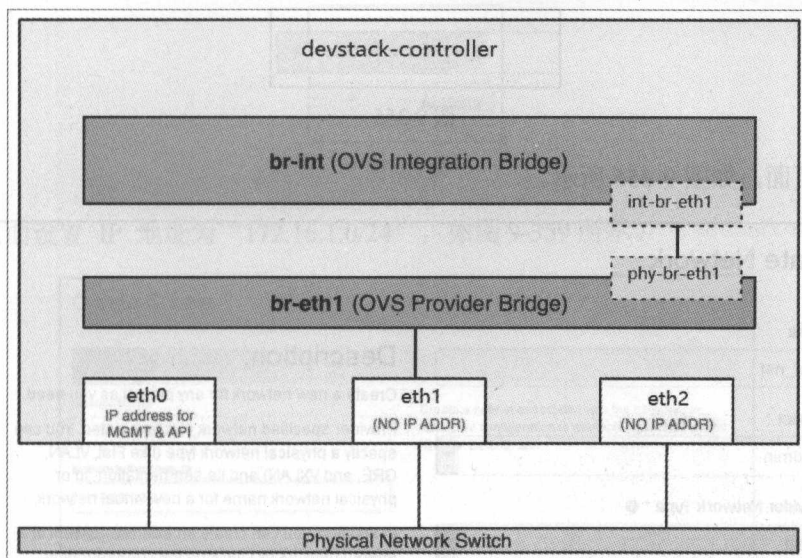


图 9-353

### veth pair VS patch port

在前面 local network 我们看到, br-int 与 Linux Bridge 之间可以通过 veth pair 连接, 如图 9-354 所示。

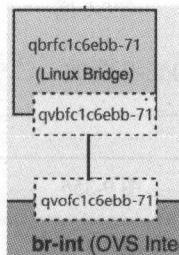


图 9-354

而这里两个 ovs bridge 之间是用 patch port 连接的。

看来 veth pair 和 patch port 都可以连接网桥, 使用的时候如何选择呢? patch port 是 ovs bridge 自己特有的 port 类型, 只能在 ovs 中使用。如果是连接两个 ovs bridge, 优先使用 patch port, 因为性能更好。所以:

- (1) 连接两个 ovs bridge, 优先使用 patch port。技术上 veth pair 也能实现, 但性能不如 patch port。
- (2) 连接 ovs bridge 和 Linux Bridge, 只能使用 veth pair。
- (3) 连接两个 Linux Bridge, 只能使用 veth pair。

### 2. 创建 flat network “flat\_net”

进入菜单 Admin → Networks, 单击 “Create Network” 按钮, 如图 9-355 所示。

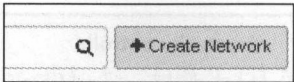


图 9-355

显示创建页面，如图 9-356 所示。

Create Network

Name

flat\_net

Project \*

admin

Provider Network Type \* ?

Flat

Physical Network \* ?

default

Admin State \*

UP

☐ Shared

☐ External Network

Description:

Create a new network for any project as you need.

Provider specified network can be created. You can specify a physical network type (like Flat, VLAN, GRE, and VXLAN) and its segmentation\_id or physical network name for a new virtual network.

In addition, you can create an external network or a shared network by checking the corresponding checkbox.

Cancel

Create Network

图 9-356

在 Provider Network Type 下拉列表中选择“Flat”。

在 Physical Network 下拉列表中填写“default”，与 ml2\_conf.ini 中 flat\_networks 参数值保持一致。

单击“Create Network”，flat\_net 创建成功，如图 9-357 所示。

<input type="checkbox"/>	Project	Network Name	Subnets Associated
<input type="checkbox"/>	admin	flat_net	
Displaying 1 item			

图 9-357

单击 flat\_net 链接，进入 network 配置页面，目前还没有 subnet，单击“Create Subnet”，如图 9-358 所示。

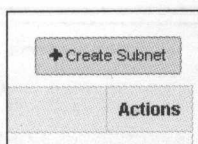


图 9-358

在创建页面设置 IP 地址为“172.16.1.0/24”，如图 9-359 所示。

**Create Subnet**

Subnet Subnet Details

**Subnet Name**  
subnet\_172\_16\_1\_0

**Network Address** ⓘ  
172.16.1.0/24

**IP Version**  
IPv4

**Gateway IP** ⓘ

☐ Disable Gateway

Create a subnet associated with the network. Advanced configuration is available by clicking on the "Subnet Details" tab.

« Back Next »

图 9-359

单击“Next”，选中“Enable DHCP”，如图 9-360 所示。

**Create Subnet**

Subnet Subnet Details

☒ Enable DHCP

Specify additional attributes for the subnet.

**Allocation Pools** ⓘ

**DNS Name Servers** ⓘ

**Host Routes** ⓘ

« Back Create »

图 9-360

单击“Create”，subnet 创建成功，如图 9-361 所示。

Subnets

<input type="checkbox"/>	Name	CIDR
<input type="checkbox"/>	subnet_172_16_1_0	172.16.1.0/24

Displaying 1 item

Ports

<input type="checkbox"/>	Name	Fixed IPs	Attached Device
<input type="checkbox"/>	(83421c44-9340)	172.16.1.2	network:dhcp

Displaying 1 item

图 9-361

(1) 底层网络发生了什么变化

查看控制节点的网络结构，执行 `ovs-vsctl show`，如图 9-362 所示。

```
root@devstack-controller:~# ovs-vsctl show
c3f35829-cb18-41e5-b2e1-d378aaa9351d
    Bridge br-ex
        Port br-ex
            Interface br-ex
                type: internal
    Bridge br-int
        fail_mode: secure
        Port br-int
            Interface br-int
                type: internal
        Port "tap83421c44-93"
            tag: 1
            Interface "tap83421c44-93"
                type: internal
        Port "int-br-eth1"
            Interface "int-br-eth1"
                type: patch
                options: {peer="phy-br-eth1"}
    Bridge "br-eth1"
        Port "phy-br-eth1"
            Interface "phy-br-eth1"
                type: patch
                options: {peer="int-br-eth1"}
        Port "eth1"
            Interface "eth1"
        Port "br-eth1"
            Interface "br-eth1"
                type: internal
```

图 9-362

Neutron 自动在 br-int 网桥上创建了 flat-net dhcp 的接口 “tap83421c44-93”。此时 flat\_net 结构如图 9-363 所示。



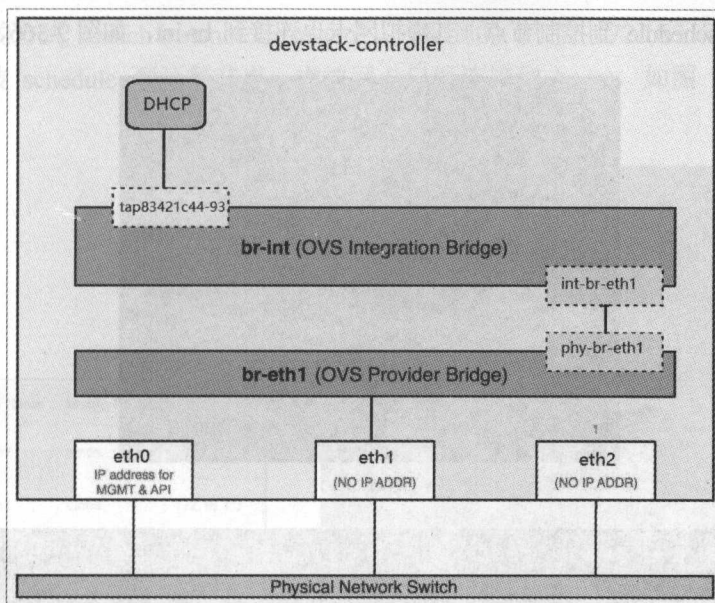


图 9-363

(2) 将 instance 连接到 flat\_net

launch 新的 instance “cirros-vm1”，网络选择 flat\_net，如图 9-364 所示。

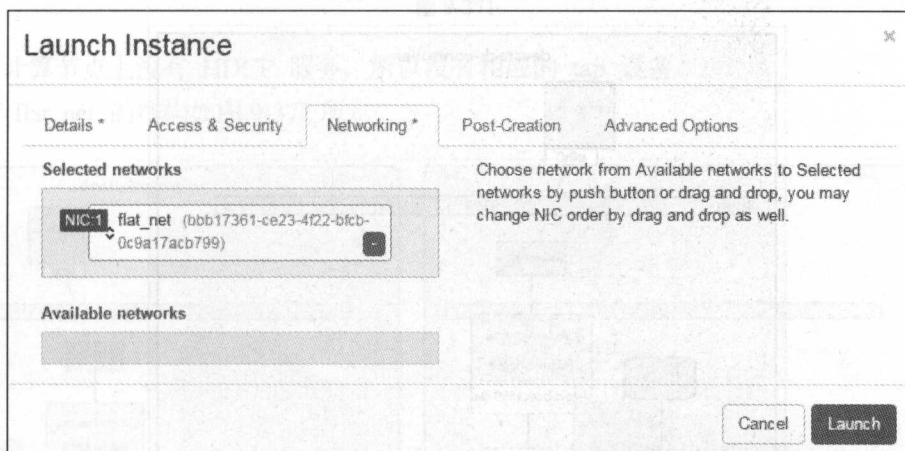


图 9-364

单击“Launch”，cirros-vm1 分配到的 IP 为 172.16.1.3，如图 9-365 所示。

<input type="checkbox"/>	Instance Name	Image Name	IP Address
<input type="checkbox"/>	cirros-vm1	cirros	172.16.1.3
Displaying 1 item			

图 9-365

ciros-vm1 被 schedule 到控制节点，其虚拟网卡也连接到 br-int，如图 9-366、图 9-367 所示。

```
root@devstack-controller:~# ovs-vsctl show
c3f35829-cb18-41e5-b2e1-d378aaa9351d
    Bridge br-ex
    Port br-ex
    Interface br-ex
        type: internal
    Bridge br-int
    fail_mode: secure
    Port "qvbaec1599-9a"
        tag: 1
        Interface "qvbaec1599-9a"
    Port br-int
    Interface br-int
        type: internal
    Port "tap83421c44-93"
        tag: 1
        Interface "tap83421c44-93"
        type: internal
    Port "int-br-eth1"
    Interface "int-br-eth1"
        type: patch
        options: {peer="phy-br-eth1"}
```

图 9-366

```
root@devstack-controller:~# brctl show
bridge name    bridge id    STP enabled  interfaces
qbrbaec1599-9a  8000.d65c6469ea5c  no          qvbaec1599-9a
                                                    tapbaec1599-9a
virbr0         8000.000000000000  yes
```

图 9-367

虚拟网卡与 br-int 的连接方式与 local 网络是一样的，不再赘述。当前 flat\_net 的结构如图 9-368 所示。

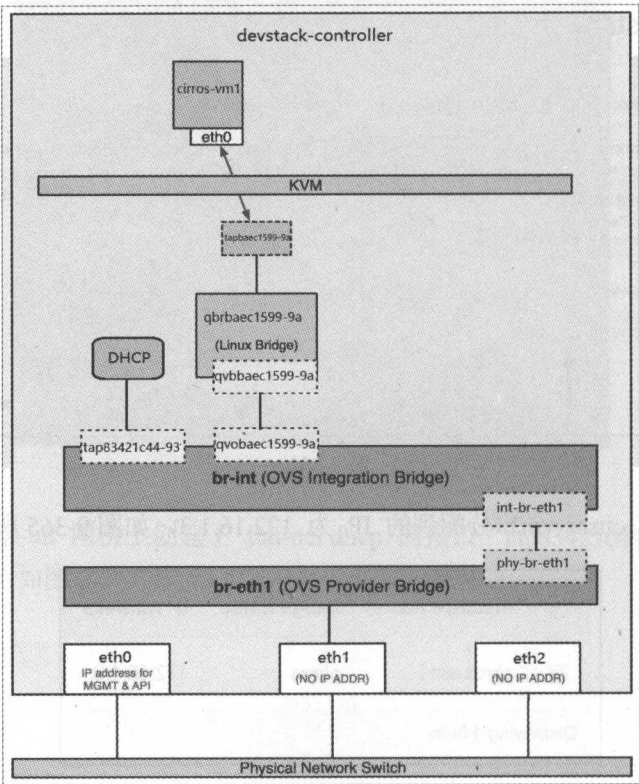


图 9-368

继续用同样的方式 launch instance cirros-vm2, 分配到的 IP 为 172.16.1.4, 如图 9-369 所示。

cirros-vm2 被 schedule 到计算节点, 虚拟网卡已经连接到 br-int, 如图 9-370、图 9-371 所示。

Instance Name	Image Name	IP Address
cirros-vm2	cirros	172.16.1.4
cirros-vm1	cirros	172.16.1.3

Displaying 2 items

图 9-369

```
root@devstack-compute1:~# ovs-vsctl show
c843c12c-9c70-4e26-99c9-ce58e3997fc6
Bridge br-int
    fail_mode: secure
    Port "qvodebf623e-de"
        tag: 1
        Interface "qvodebf623e-de"
    Port "int-br-eth1"
        Interface "int-br-eth1"
        type: patch
        options: {peer="phy-br-eth1"}
    Port br-int
        Interface br-int
        type: internal
Bridge "br-eth1"
    Port "eth1"
        Interface "eth1"
    Port "br-eth1"
        Interface "br-eth1"
        type: internal
    Port "phy-br-eth1"
        Interface "phy-br-eth1"
        type: patch
        options: {peer="int-br-eth1"}
```

图 9-370

```
root@devstack-compute1:~# brctl show
bridge name      bridge id      STP enabled  interfaces
qbrdebf623e-de   8000.86b26ea5b937  no          qybdebf623e-de
virbr0           8000.000000000000  yes         tapdebf623e-de
root@devstack-compute1:~#
```

图 9-371

因为计算节点上没有 HDCP 服务, 所以没有相应的 tap 设备。

当前 flat\_net 的结构如图 9-372 所示。

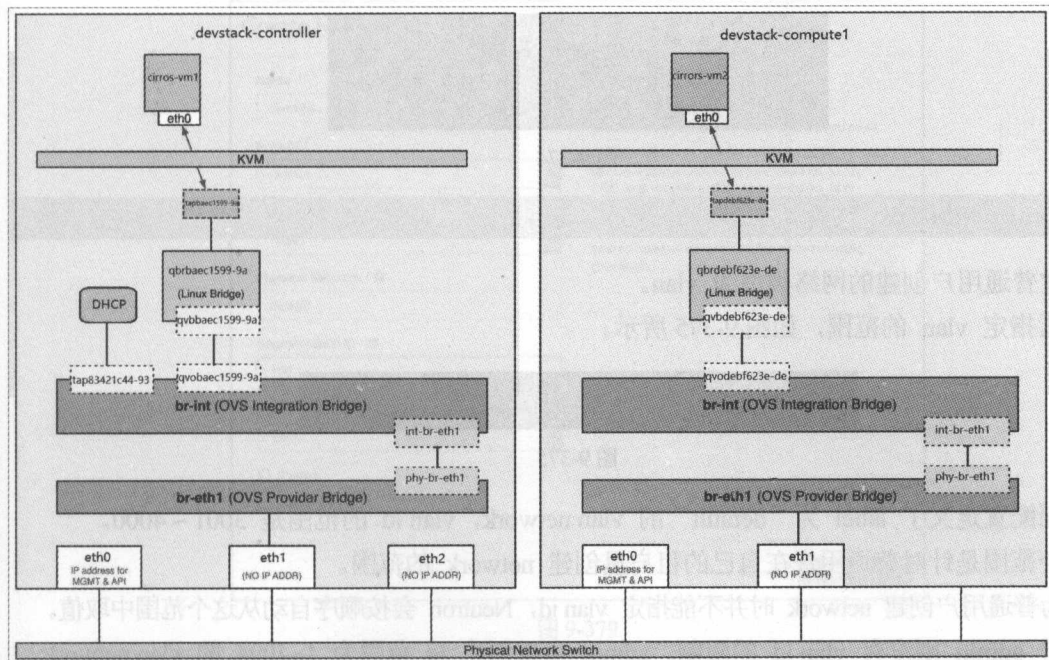


图 9-372

示 cirros-vm1 (172.16.1.3) 与 cirros-vm2 (172.16.1.4) 位于不同节点, 通过 flat\_net 相连, 下面验证连通性。

在 cirros-vm2 控制台中 ping 172.16.1.3, 如图 9-373 所示。

```
$ hostname
cirros-vm2
$ ping 172.16.1.3
PING 172.16.1.3 (172.16.1.3): 56 data bytes
64 bytes from 172.16.1.3: seq=0 ttl=64 time=2.640 ms
64 bytes from 172.16.1.3: seq=1 ttl=64 time=1.986 ms
64 bytes from 172.16.1.3: seq=2 ttl=64 time=1.620 ms
64 bytes from 172.16.1.3: seq=3 ttl=64 time=2.221 ms
64 bytes from 172.16.1.3: seq=4 ttl=64 time=1.640 ms
```

图 9-373

如我们预料, ping 成功。

## 9.5.7 vlan network

vlan network 是带 tag 的网络。在 Open vSwitch 实现方式下, 不同 vlan instance 的虚拟网卡都接到 br-int 上。这一点与 Linux Bridge 非常不同, Linux Bridge 是不同 vlan 接到不同的网桥上。

在我们的实验环境中, 收发 vlan 数据的物理网卡为 eth1, 上面可以走多个 vlan, 所以物理交换机上与 eth1 相连的 port 要设置成 trunk 模式, 而不是 access 模式。

### 1. 在 ML2 配置中 enable vlan network

在 /etc/neutron/plugins/ml2/ml2\_conf.ini 设置 vlan network 相关参数, 如图 9-374 所示。

```
tenant_network_types = vlan
extension_drivers = port_security
type_drivers = local,flat,vlan,gre,vxlan
mechanism_drivers = openvswitch
```

图 9-374

```
tenant_network_types = vlan
```

指定普通用户创建的网络类型为 vlan。

然后指定 vlan 的范围, 如图 9-375 所示。

```
network_vlan_ranges = default:
```

图 9-375

上面配置定义了 label 为 “default” 的 vlan network, vlan id 的范围是 3001~4000。

这个范围是针对普通用户在自己的租户里创建 network 的范围。

因为普通用户创建 network 时并不能指定 vlan id, Neutron 会按顺序自动从这个范围中取值。

对于 admin 则没有 vlan id 的限制, admin 可以创建 id 范围为 1~4094 的 vlan network。

接着需要指明 vlan 网络与物理网络的对应关系, 如图 9-376 所示。



```
bridge_mappings: = default:br-eth1
```

图 9-376

如图 9-376 所示，在 [ml2\_type\_vlan] 中定义了 label “default”，[ovs] 中则通过 bridge\_mappings 指明 default 对应的 Open vSwitch 网桥为 br-eth1。

这里 label 的作用与前面 flat network 中的 label 一样，只是一个标示，可以是任何字符串。我们需要提前通过 ovs-ovctl 命令：

- 创建 br-eth1。
- 将物理网卡 eth1 桥接在 br-eth1 上，如图 9-377 所示。

```
root@devstack-controller:~# ovs-vsctl add-br br-eth1
root@devstack-controller:~# ovs-vsctl add-port br-eth1 eth1
root@devstack-controller:~#
```

图 9-377

## 2. 创建第一个 vlan network“vlan100”

打开菜单 Admin → Networks，单击“Create Network”按钮，如图 9-378 所示。

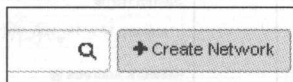


图 9-378

显示创建页面，如图 9-379 所示。

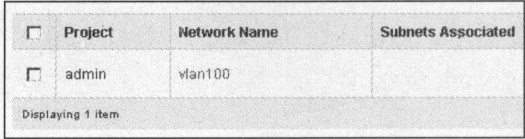
图 9-379

在 Provider Network Type 下拉列表中选择“VLAN”。

在 Physical Network 下拉列表中填写“default”，与 ml2\_conf.ini 中 network\_vlan\_ranges 参数数值保持一致。

Segmentation ID 即 VLAN ID，设置为 100。

单击“Create Network”，vlan100 创建成功，如图 9-380 所示。



<input type="checkbox"/>	Project	Network Name	Subnets Associated
<input type="checkbox"/>	admin	vlan100	

Displaying 1 item

图 9-380

单击 vlan100 链接，进入 network 配置页面，目前还没有 subnet，单击“Create Subnet”按钮，如图 9-381 所示。

创建 subnet\_172\_16\_100\_0，IP 地址为 172.16.100.0/24，如图 9-382、图 9-383 所示。

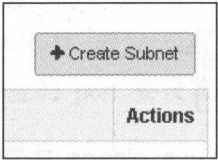
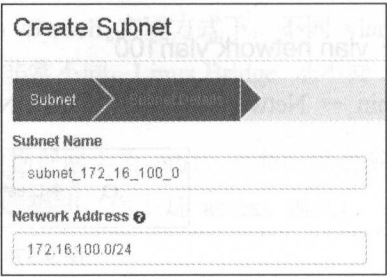


图 9-381



Create Subnet

Subnet > Subnet ID

Subnet Name



subnet\_172\_16\_100\_0

Network Address

172.16.100.0/24



图 9-382

## Subnets

	Name	CIDR
	<u>subnet_172_16_100_0</u>	172.16.100.0/24

Displaying 1 item

## Ports

	Name	Fixed IPs	Attached Device
	(43567363-5096)	172.16.100.2	network:dhcp

Displaying 1 item

图 9-383

(1) 底层网络发生了什么变化

在控制节点上执行 ovs-vsctl show，查看网络结构，如图 9-384 所示。

```

root@devstack-controller:~# ovs-vsctl show
c3f35829-cb18-41e5-b2e1-d378aaa9351d
    Bridge br-ex
        Port br-ex
            Interface br-ex
                type: internal
    Bridge br-int
        fail_mode: secure
        Port br-int
            Interface br-int
                type: internal
        Port "tap43567363-50"
            tag: 1
            Interface "tap43567363-50"
                type: internal
        Port "int-br-eth1"
            Interface "int-br-eth1"
                type: patch
                options: {peer="phy-br-eth1"}
    Bridge "br-eth1"
        Port "phy-br-eth1"
            Interface "phy-br-eth1"
                type: patch
                options: {peer="int-br-eth1"}
        Port "eth1"
            Interface "eth1"
        Port "br-eth1"
            Interface "br-eth1"
                type: internal

```

图 9-384

Neutron 自动在 br-int 网桥上创建了 vlan100 DHCP 的接口 “tap43567363-50”。此时 vlan100 结构如图 9-385 所示。

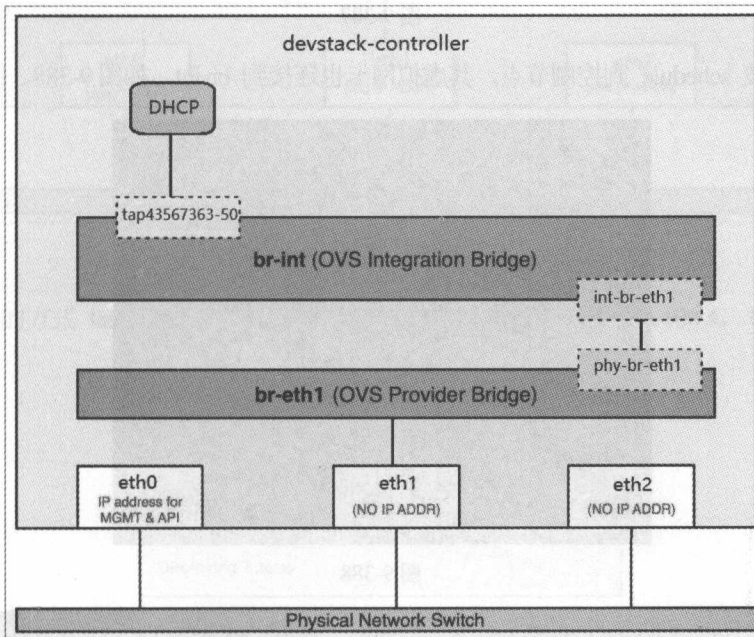


图 9-385

(2) 将 instance 连接到 vlan100

launch 新的 instance “cirros-vm1”，网络选择 vlan100，如图 9-386 所示。

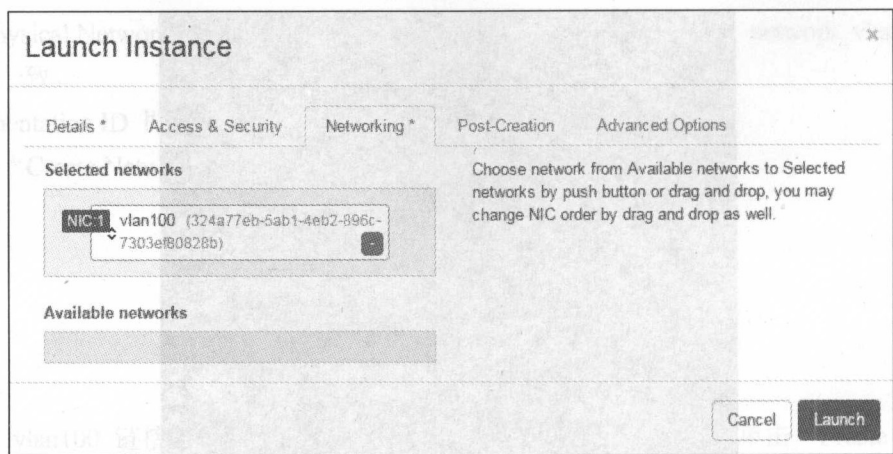


图 9-386

cirros-vm1 分配到的 IP 为 172.16.100.3, 如图 9-387 所示。

	Instance Name	Image Name	IP Address
<input checked="" type="checkbox"/>	cirros-vm1	cirros	172.16.100.3

Displaying 1 item

图 9-387

cirros-vm1 被 schedule 到控制节点, 其虚拟网卡也连接到 br-int, 如图 9-388、图 9-389 所示。

```
root@devstack-controller:~# ovs-vsctl show
c3f35829-cb18-41e5-b2e1-d378aaa9351d.
  Bridge br-ex
    Port br-ex
      Interface br-ex
        type: internal
  Bridge br-int
    fail_mode: secure
    Port br-int
      Interface br-int
        type: internal
    Port "tap43567363-50"
      tag: 1
      Interface "tap43567363-50"
        type: internal
    Port "qvo1fb4cdbf-8f"
      tag: 1
      Interface "qvo1fb4cdbf-8f"
    Port "int-br-eth1"
      Interface "int-br-eth1"
        type: patch
        options: {peer="phy-br-eth1"}
```

图 9-388

bridge name	bridge id	STP enabled	interfaces
qbr1fb4cdbf-8f	8000.e684bf50ed34	no	qvb1fb4cdbf-8f
virbr0	8000.000000000000	yes	tap1fb4cdbf-8f

图 9-389

虚拟网卡与 br-int 的连接方式与 local 和 flat 网络没有任何区别, 不再赘述。

当前 vlan100 的结构, 如图 9-390 所示。



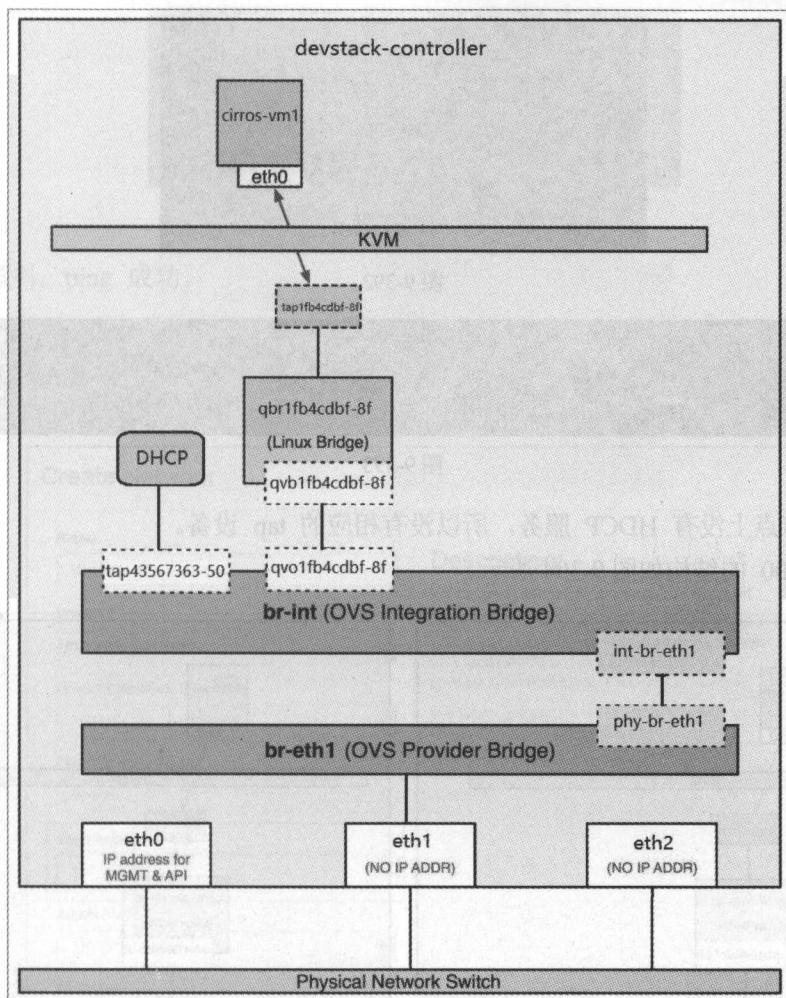


图 9-390

继续用同样的方式 launch instance cirros-vm2, 分配到的 IP 为 172.16.100.4, 如图 9-391 所示。

<input type="checkbox"/>	Instance Name	Image Name	IP Address
<input type="checkbox"/>	cirros-vm2	cirros	172.16.100.4
<input type="checkbox"/>	cirros-vm1	cirros	172.16.100.3
Displaying 2 items			

图 9-391

cirros-vm2 被 schedule 到计算节点, 虚拟网卡已经连接到 br-int, 如图 9-392、图 9-393 所示。

```
root@devstack-compute1:~# ovs-vsctl show
c843c12c-9c70-4e26-99c9-ce58e3997fc6
Bridge br-int
  fail_mode: secure
  Port "int-br-eth1"
    Interface "int-br-eth1"
      type: patch
      options: {peer="phy-br-eth1"}
  Port br-int
    Interface br-int
      type: internal
  Port "qvo4139d09b-30"
    tag: 2
    Interface "qvo4139d09b-30"
```

图 9-392

```
root@devstack-compute1:~# brctl show
bridge name      bridge id        STP enabled  interfaces
qbr4139d09b-30   8000.3a5a32b3f86c  no          qvb4139d09b-30
                tap4139d09b-30
virbr0           8000.000000000000  yes
```

图 9-393

因为计算节点上没有 HDCP 服务，所以没有相应的 tap 设备。  
当前 vlan100 的结构如图 9-394 所示。

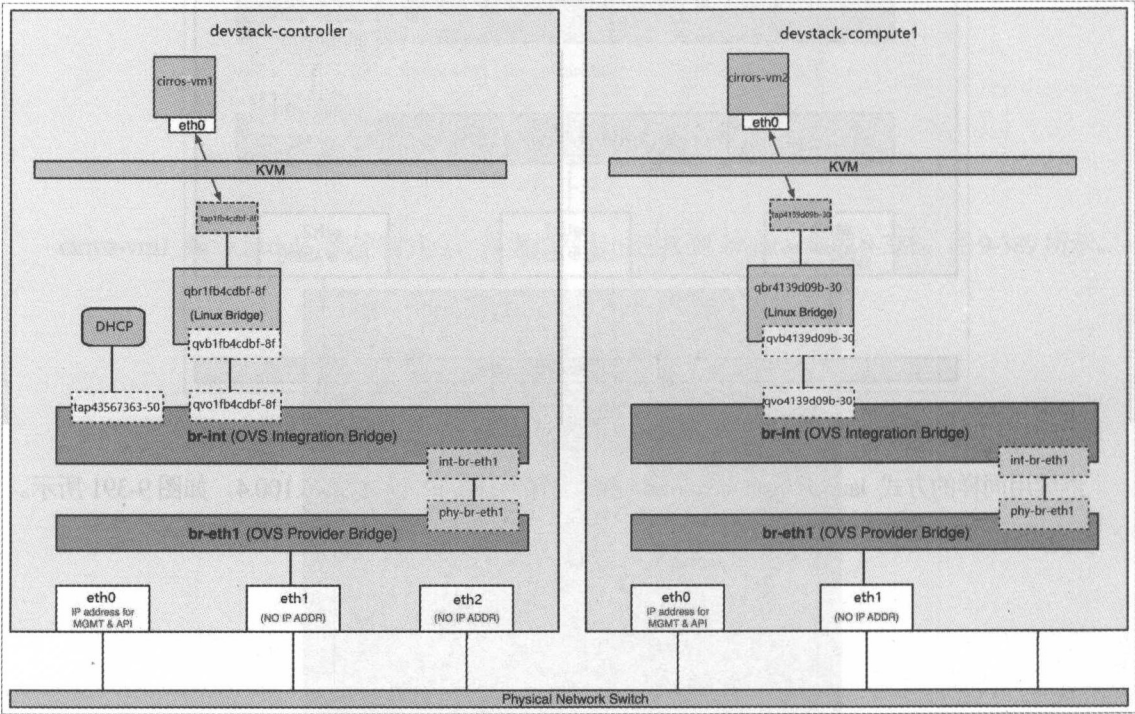


图 9-394

ciros-vm1 (172.16.100.3) 与 ciros-vm2 (172.16.100.4) 位于不同节点，通过 vlan100 相连，下面执行 PING 验证连通性。

在 ciros-vm1 控制台中，执行 ping 172.16.100.4，如图 9-395 所示。

```

$ hostname
cirros-vm1
$
$ ping 172.16.100.4
PING 172.16.100.4 (172.16.100.4): 56 data bytes
64 bytes from 172.16.100.4: seq=0 ttl=64 time=3.260 ms
64 bytes from 172.16.100.4: seq=1 ttl=64 time=1.951 ms
64 bytes from 172.16.100.4: seq=2 ttl=64 time=2.520 ms
64 bytes from 172.16.100.4: seq=3 ttl=64 time=1.800 ms
64 bytes from 172.16.100.4: seq=4 ttl=64 time=1.086 ms

```

图 9-395

如我们预料, ping 成功。

### 3. 创建第二个 valn network“vlan101”

继续创建第二个 vlan network “vlan101”，如图 9-396 所示。

图 9-396

subnet IP 地址为 172.16.101.0/24，如图 9-397 所示。

<input type="checkbox"/>	Project	Network Name	Subnets Associated
<input type="checkbox"/>	admin	vlan101	subnet_172_16_101_0 172.16.101.0/24
<input type="checkbox"/>	admin	vlan100	subnet_172_16_100_0 172.16.100.0/24

Displaying 2 items

图 9-397

#### (1) 底层网络发生了什么变化

Neutron 自动在 br-int 网桥上创建了 vlan100 DHCP 的接口 “tap1820558c-0a”，如图 9-398 所示。

```
root@devstack-controller:~# ovs-vsctl show
c3f35829-cb18-41e5-b2e1-d378aaa9351d
    Bridge br-ex
        Port br-ex
            Interface br-ex
                type: internal
    Bridge br-int
        fail_mode: secure
        Port br-int
            Interface br-int
                type: internal
    Port "tap1820558c-0a"
        tag: 2
        Interface "tap1820558c-0a"
            type: internal
    Port "tap43567363-50"
        tag: 1
        Interface "tap43567363-50"
            type: internal
    Port "qvo1fb4cdbf-8f"
        tag: 1
        Interface "qvo1fb4cdbf-8f"
    Port "int-br-eth1"
        Interface "int-br-eth1"
            type: patch
            options: {peer="phy-br-eth1"}
    Create Node {}
```

图 9-398

现在，网络结构如图 9-399 所示。

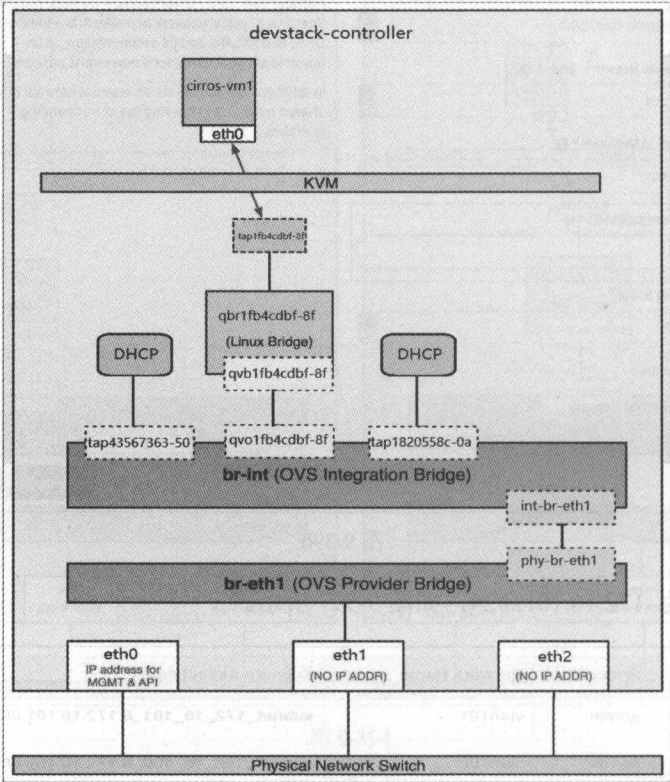


图 9-399

(2) 将 instance 连接到 vlan101  
launch 新的 instance “cirros-vm3”，网络选择 vlan101，如图 9-400 所示。



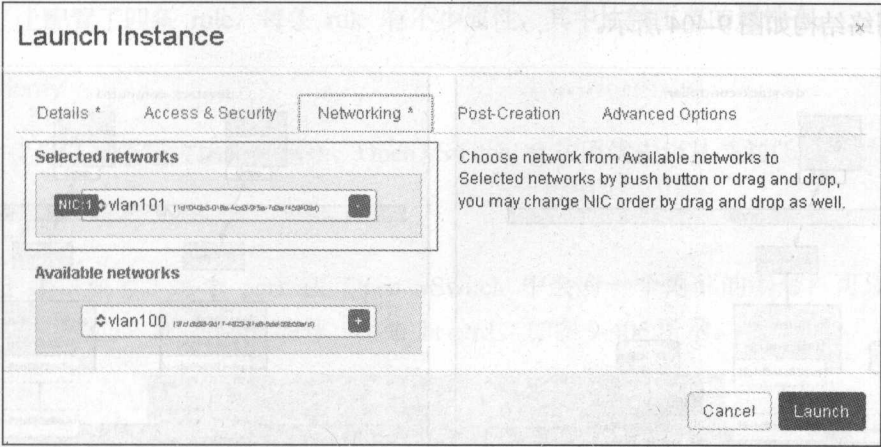


图 9-400

cirros-vm3 分配到的 IP 为 172.16.101.103, 如图 9-401 所示。

<input type="checkbox"/>	Instance Name	Image Name	IP Address
<input type="checkbox"/>	cirros-vm3	cirros	172.16.101.3
<input type="checkbox"/>	cirros-vm2	cirros	172.16.100.4
<input type="checkbox"/>	cirros-vm1	cirros	172.16.100.3
Displaying 3 items			

图 9-401

cirros-vm3 被 schedule 到计算节点, 虚拟网卡已经连接到 br-int, 如图 9-402、图 9-403 所示。

```
root@devstack-computel:~# ovs-vsctl show
c843c12c-9c70-4e26-99c9-ce58e3997fc6
    Bridge br-int
        fail_mode: secure
        Port "qvo98582dc9-db"
            tag: 5
            Interface "qvo98582dc9-db"
        Port "int-br-eth1"
            Interface "int-br-eth1"
            type: patch
            options: {peer="phy-br-eth1"}
        Port "qvo4139d09b-30"
            tag: 1
            Interface "qvo4139d09b-30"
        Port br-int
            Interface br-int
            type: internal
```

图 9-402

```
root@devstack-computel:~# brctl show
Bridge name      bridge id      STP enabled    interfaces
qbr4139d09b-30    8000.0e18bc5c5c3f    no             qvb4139d09b-30
qbr98582dc9-db    8000.3281e0f1091b    no             tap4139d09b-30
vibr0             8000.000000000000    yes            qvb98582dc9-db
tap98582dc9-db    8000.000000000000    no             tap4139d09b-30
```

图 9-403

当前网络结构如图 9-404 所示。

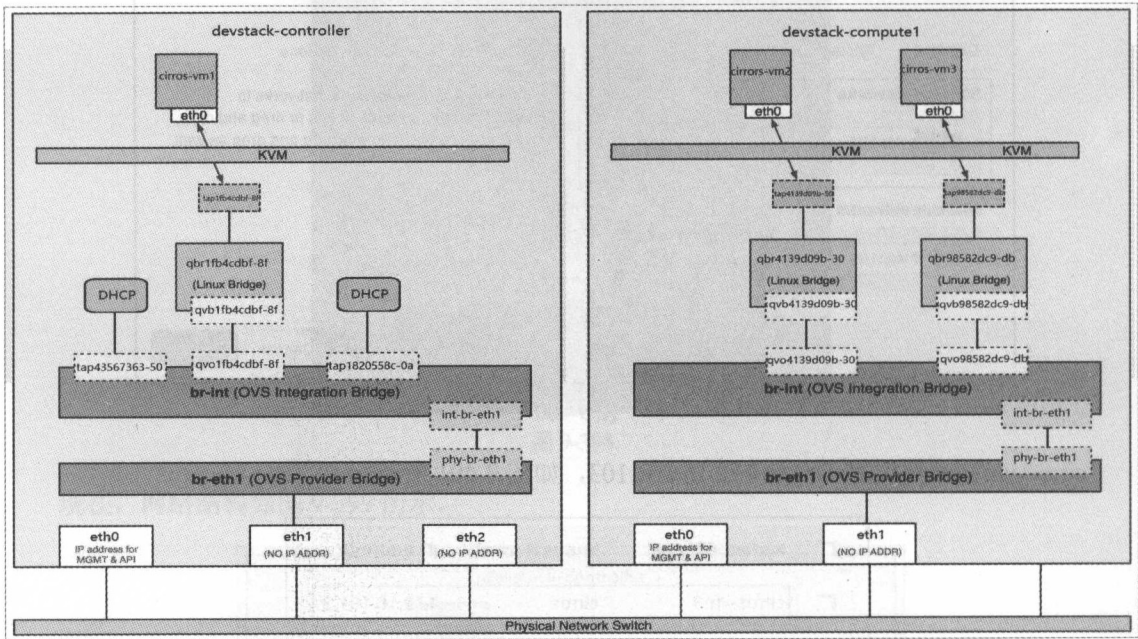


图 9-404

- cirros-vm1 位于控制节点，属于 vlan100。
- cirros-vm2 位于计算节点，属于 vlan100。
- cirros-vm3 位于计算节点，属于 vlan101。
- cirros-vm1 与 cirros-vm2 都在 vlan100，它们之间能通信。
- cirros-vm3 在 vlan101，不能与 cirros-vm1 和 cirros-vm2 通信。

上面的结论是毋庸置疑的，但我们更需要关心的是：Open vSwitch 是如何实现 vlan100 和 vlan101 的隔离？

与 Linux Bridge driver 不同，Open vSwitch driver 并不通过 eth1.100、eth1.101 等 VLAN interface 来隔离不同的 VLAN。所有的 instance 都连接到同一个网桥 br-int，Open vSwitch 通过 flow rule（流规则）来指定如何对进出 br-int 的数据进行转发。

当数据进出 br-int 时，flow rule 可以修改、添加或者剥掉数据包包的 VLAN tag。Neutron 负责创建 flow rule 并将它们配置到 br-int、br-eth1 等 Open vSwitch 上。

下面我们就来研究一下当前的 flow rule。

查看 flow rule 的命令是 ovs-ofctl dump-flow。

首先查看计算节点 br-eth1 的 flow rule，如图 9-405 所示。

```
root@devstack-compute1:~# ovs-ofctl dump-flows br-eth1
NXST:Flow reply (xid=0x1):
cookie=0x0, duration=35691.011s, table=0, n_packets=12, n_bytes=2362, idle_age=349s, priority=2, in_port=2, dl_vlan=100, actions=mod_vlan_vid=100, cookie=0x0, duration=35691.011s, table=0, n_packets=41, n_bytes=3576, idle_age=350s, priority=2, in_port=2, dl_vlan=101, actions=mod_vlan_vid=101, cookie=0x0, duration=35691.523s, table=0, n_packets=10, n_bytes=2352, idle_age=3311s, priority=2, in_port=2, actions=mod_vlan_vid=101, cookie=0x0, duration=35691.556s, table=0, n_packets=22802696, n_bytes=1467532296, idle_age=0, priority=0, actions=normal
root@devstack-compute1:~#
```

图 9-405

br-eth1 上配置了四条 rule，每条 rule 有不少属性，其中比较重要的属性有：

- priority

rule 的优先级，值越大优先级越高。Open vSwitch 会按照优先级从高到低应用规则。

- in\_port

inbound 端口编号，每个 port 在 Open vSwitch 中会有一个内部的编号。可以通过命令 `ovs-ofctl show <bridge>` 查看 port 编号。比如 br-eth1，如图 9-406 所示。

```
root@devstack-compute1:~# ovs-ofctl show br-eth1
OFPT_FEATURES_REPLY (xid=0x2): dpid:0000005056b0b540
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS
actions: OUTPUT SET_VLAN_VID SET_VLAN_PCP STRIP_VLAN
1(eth1): addr:00:50:56:b0:b5:40
  config: 0
  state: 0
  current: 10GB-FD COPPER
  advertised: COPPER
  supported: 1GB-FD 10GB-FD COPPER
  speed: 10000 Mbps now, 10000 Mbps max
2(phy-br-eth1): addr:ba:2b:50:90:4b:44
  config: 0
  state: 0
  speed: 0 Mbps now, 0 Mbps max
LOCAL(br-eth1): addr:e2:99:8b:b8:56:41
  config: PORT_DOWN
  state: LINK_DOWN
  speed: 0 Mbps now, 0 Mbps max
```

图 9-406

eth1 编号为 1，phy-br-eth1 编号为 2。

- dl\_vlan

数据包原始的 VLAN ID。

- actions

对数据包进行的操作。

br-eth1 跟 VLAN 相关的 flow rule 是前面两条，下面我们来详细分析。

为了清晰起见，我们只保留重要的信息，如下：

```
priority=4,in_port=2,dl_vlan=1 actions=mod_vlan_vid:100,NORMAL
priority=4,in_port=2,dl_vlan=5 actions=mod_vlan_vid:101,NORMAL
```

第一条的含义是：从 br-eth1 的端口 phy-br-eth1 (in\_port=2) 接收进来的包，如果 VLAN ID 是 1 (dl\_vlan=1)，那么需要将 VLAN ID 改为 100 (actions=mod\_vlan\_vid:100)，如图 9-407 所示。

从上面的网络结构我们可知，phy-br-eth1 连接的是 br-int，phy-br-eth1 的 inbound 包实际上就是 instance 通过 br-int 发送给物理网卡的数据。

那么怎么理解将 VLAN ID 1 改为 VLAN ID 100 呢？

请看下面计算节点 `ovs-vsctl show` 的输出，如图 9-408 所示。

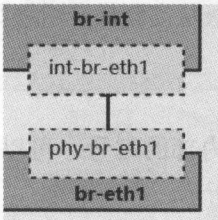


图 9-407

```
root@devstack-comput1:~# ovs-vsctl show
c843c12c-9c70-4e26-99c9-ce58e3997fc6
    Bridge br-int
    fail_mode: secure
    Port "qvo98582dc9-db"
        tag: 5
        Interface "qvo98582dc9-db"
    Port "int-br-eth1"
        Interface "int-br-eth1"
        type: patch
        options: {peer="phy-br-eth1"}
    Port "qvo4139d09b-30"
        tag: 1
        Interface "qvo4139d09b-30"
    Port br-int
        Interface br-int
        type: internal
```

图 9-408

br-int 通过 tag 隔离不同的 port，这个 tag 可以看成内部的 VLAN ID。

从 qvo4139d09b-30（对应 cirros-vm2，vlan100）进入的数据包会被打上 1 的 VLAN tag。

从 qvo98582dc9-db（对应 cirros-vm3，vlan101）进入的数据包会被打上 5 的 VLAN tag。

因为 br-int 中的 VLAN ID 跟物理网络中的 VLAN ID 并不相同，所以当 br-eth1 接收到 br-int 发来的数据包时，需要对 VLAN 就行转换。Neutron 负责维护 VLAN ID 的对应关系，并将转换规则配置在 flow rule 中。

理解了 br-eth1 的 flow rule，我们再来分析 br-int 的 flow rule，如图 9-409 所示。

```
root@devstack-comput1:~# ovs-ofctl dump-flows br-int
NXY:Flow reply (xid=0x4):
cookie=0x98576e5eb0e6b0, duration=38842.511s, table=0, n_packets=0, n_bytes=0, idle_age=38842, priority=10,icmp6,in_port=2,icmp_type=130 actions=resubmit(,24)
cookie=0x09da760e5eb0e6b0, duration=38842.409s, table=0, n_packets=0, n_bytes=0, idle_age=38842, priority=10,icmp6,in_port=6,icmp_type=130 actions=resubmit(,24)
cookie=0x09da760e5eb0e6b0, duration=38842.45s, table=0, n_packets=18, n_bytes=18, idle_age=414, priority=3,in_port=1,d_l_vlan=100 actions=mod_vlan_vid:1,NORMAL
cookie=0x09da760e5eb0e6b0, duration=38842.475s, table=0, n_packets=8389, n_bytes=800930, idle_age=0, priority=3,in_port=1,d_l_vlan=100 actions=mod_vlan_vid:1,NORMAL
cookie=0x09da760e5eb0e6b0, duration=38842.5s, table=0, n_packets=11, n_bytes=462, idle_age=38843, priority=10,arp,in_port=2,arp_spa=172.16.101.3 actions=resubmit(,24)
cookie=0x09da760e5eb0e6b0, duration=38842.505s, table=0, n_packets=21, n_bytes=882, idle_age=414, priority=2,arp,in_port=6,arp_spa=172.16.101.3 actions=resubmit(,24)
cookie=0x09da760e5eb0e6b0, duration=38842.005s, table=0, n_packets=125294, n_bytes=99204175, idle_age=0, priority=2,in_port=1 actions=drop
cookie=0x09da760e5eb0e6b0, duration=38842.126s, table=0, n_packets=244, n_bytes=20041, idle_age=4126, priority=0 actions=NORMAL
cookie=0x09da760e5eb0e6b0, duration=38842.123s, table=23, n_packets=0, n_bytes=0, idle_age=38842, priority=0 actions=drop
cookie=0x09da760e5eb0e6b0, duration=38842.516s, table=24, n_packets=0, n_bytes=0, idle_age=38842, priority=0,icmp6,in_port=2,icmp_type=136,nd_target=f80:f816:3eff:f816:3:3 actions=NORMAL
cookie=0x09da760e5eb0e6b0, duration=38842.414s, table=24, n_packets=0, n_bytes=0, idle_age=38842, priority=2,icmp6,in_port=6,icmp_type=136,nd_target=f80:f816:3eff:f816:3:3 actions=NORMAL
cookie=0x09da760e5eb0e6b0, duration=38842.404s, table=24, n_packets=21, n_bytes=882, idle_age=4114, priority=2,arp,in_port=6,arp_spa=172.16.101.3 actions=NORMAL
cookie=0x09da760e5eb0e6b0, duration=38842.506s, table=24, n_packets=11, n_bytes=462, idle_age=38648, priority=2,arp,in_port=2,arp_spa=172.16.100.4 actions=NORMAL
cookie=0x09da760e5eb0e6b0, duration=38842.116s, table=24, n_packets=0, n_bytes=0, idle_age=38842, priority=0 actions=drop
```

图 9-409

最关键的是下面两条：

```
priority=3,in_port=1,d_l_vlan=100 actions=mod_vlan_vid:1,NORMAL
priority=3,in_port=1,d_l_vlan=101 actions=mod_vlan_vid:5,NORMAL
```

port 1 为 int-br-eth1，如图 9-410 所示。那么这两条规则的含义就应该是：

- (1) 从物理网卡接收进来的数据包，如果 VLAN 为 100，则改为内部 VLAN 1。
- (2) 从物理网卡接收进来的数据包，如果 VLAN 为 101，则将为内部 VLAN 5。

```
root@devstack-comput1:~# ovs-ofctl show br-int
OFPT_FEATURES_REPLY (xid=0x2): dpid:0000a6cfb820
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS
actions: OUTPUT SET_VLAN_VID SET_VLAN_PCP STRIP_VLAN
1(int-br-eth1): addr:76:b2:82:e2:10:2a
    config: 0
    state: 0
    speed: 0 Mbps now, 0 Mbps max
2(qvo4139d09b-30): addr:72:f2:f2:f8:2b:f2
```

图 9-410



简单地说,数据包在物理网络中通过 VLAN 100 和 VLAN 101 隔离,在计算节点的 br-int 中则是通过内部 VLAN 1 和 VLAN 5 隔离。

控制节点的 flow rule 非常类似,留给大家分析。

## 9.5.8 Routing

路由服务提供跨 subnet 互联互通的能力。

例如,前面我们搭建了实验环境:

```

cirros-vm1      172.16.100.3      vlan100
cirros-vm3      172.16.101.3      vlan101
  
```

这两个 instance 要通信必须借助 router,可以是物理 router 或者虚拟 router。

下面详细讨论 Neutron 的虚拟 router 实现。

### 1. 配置 L3 Agent

Neutron 的路由服务是由 L3 Agent 提供的。L3 Agent 需要正确配置才能工作,配置文件为 /etc/neutron/l3\_agent.ini,位于控制节点或网络节点,如图 9-411 所示。

```

[DEFAULT]
l3_agent_manager = neutron.agent.l3_agent.L3NATAgentwithStateReport
external_network_bridge = br-ex
interface_driver = neutron.agent.linux.interface.OVSInterfaceDriver
ovs_user_veth = False
use_namespaces = True
debug = True
verbose = True
  
```

图 9-411

devstack 已经帮我们配置默认的属性,大部分情况下不需要修改就可以使用。

external\_network\_bridge 指定连接外网的网桥,默认是 br-ex。

interface\_driver 是最重要的选项,如果 mechanism driver 是 open vswitch,则:

```
interface_driver=neutron.agent.linux.interface.OVSInterfaceDriver
```

如果选用 Linux Bridge,则:

```
interface_driver=neutron.agent.linux.interface.BridgeInterfaceDriver
```

L3 Agent 运行在控制或网络节点,如图 9-412 所示。

```

root@devstack-controller:~# neutron agent-list
  
```

id	agent_type	host	alive
24167538-1e0b-4417-acd2-0956fe116266	Metadata agent	devstack-controller	--
430c8c90-ef04-4bce-87f9-0dd861e84001	Open vswitch agent	devstack-compute1	--
6ff2380a-73df-4b11-b40e-a08dd20fedbd	Loadbalancer agent	devstack-controller	--
7cd22aa9-9e47-4d6d-b9fe-36d6ce4a0655	DHCP agent	devstack-controller	--
a88ff4f9-b0e9-47ef-a1b2-6134190da7b2	Open vswitch agent	devstack-controller	--
ac48a3dc-e591-4489-b191-c467383a398f	L3 agent	devstack-controller	--

图 9-412

2. 用虚拟 router 实现 subnet 间路由

下面将创建虚拟 router “router\_100\_101”，打通 vlan100 和 vlan101。  
创建 router。进入操作菜单 Project → Network → Routers，如图 9-413 所示。  
单击 “Create Router”，如图 9-414 所示。

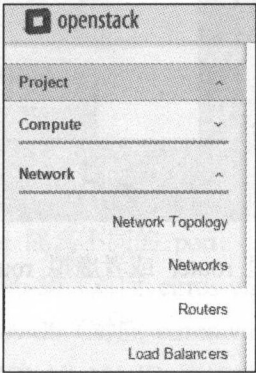


图 9-413

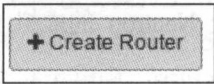


图 9-414

router 命名为 “router\_100\_101”，单击 “Create Router” 确认，如图 9-415 所示。

Create Router

Router Name \*

router\_100\_101

Admin State

UP

Description:

Creates a router with specified parameters.

Cancel

Create Router

图 9-415

router\_100\_101 创建成功，如图 9-416 所示。

Routers		
<input type="checkbox"/>	Name	Status
<input type="checkbox"/>	router_100_101	Active
Displaying 1 item		

图 9-416

接下来需要将 vlan100 和 vlan101 连接到 router\_100\_101。

单击“router\_100\_101”链接进入 router 的配置页面，在“Interfaces”标签中单击“Add Interface”，如图 9-417 所示。

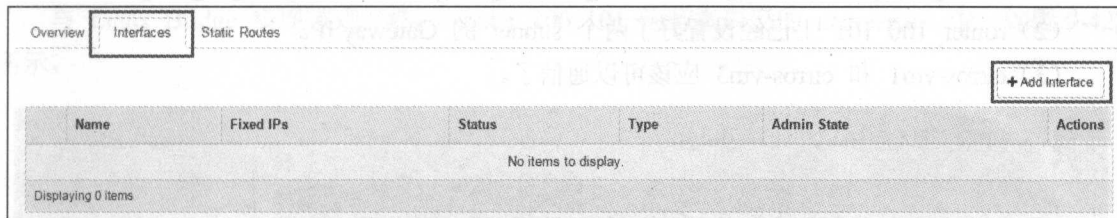


图 9-417

选择 vlan100 的 subnet\_172\_16\_100\_0，单击“Add Interface”确认，如图 9-418 所示。

Add Interface

Subnet \*

vlan100: 172.16.100.0/24 (subnet\_172\_16\_100)

IP Address (optional) ?

Router Name \*

router\_100\_101

Router ID \*

a81cc110-16f4-4d6c-89d2-8af91cec9714

Description:

You can connect a specified subnet to the router.  
The default IP address of the interface created is a gateway of the selected subnet. You can specify another IP address of the interface here. You must select a subnet to which the specified IP address belongs to from the above list.

Cancel

Add interface

图 9-418

用同样的方法添加 vlan101 的 subnet\_172\_16\_101\_0。

完成后，可以看到 router\_100\_101 有了两个 interface，其 IP 正好是 subnet 的 Gateway IP 172.16.100.1 和 172.16.101.1，如图 9-419 所示。

Overview Interfaces Static Routes			
<input type="checkbox"/>	Name	Fixed IPs	Status
<input type="checkbox"/>	(2ffdb861-731c)	172.16.100.1	Active
<input type="checkbox"/>	(d295b258-4586)	172.16.101.1	Active
Displaying 2 items			

图 9-419

到这里，我们可以预见：

- (1) router\_100\_101 已经连接了 subnet\_172\_16\_100\_0 和 subnet\_172\_16\_101\_0。
- (2) router\_100\_101 上已经设置好了两个 subnet 的 Gateway IP。
- (3) cirros-vm1 和 cirros-vm3 应该可以通信了。

通过 PING 测试一下，如图 9-420 所示。

```
$ hostname
cirros-vm1
$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        inet6 ::1/128 scope host
            valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
    link/ether fa:16:3e:8f:31:a3 brd ff:ff:ff:ff:ff:ff
    inet 172.16.100.3/24 brd 172.16.100.255 scope global eth0
        inet6 fe80::f816:3eff:fe8f:31a3/64 scope link
            valid_lft forever preferred_lft forever
$ ping 172.16.101.3
PING 172.16.101.3 (172.16.101.3): 56 data bytes
64 bytes from 172.16.101.3: seq=0 ttl=63 time=3.111 ms
64 bytes from 172.16.101.3: seq=1 ttl=63 time=1.808 ms
64 bytes from 172.16.101.3: seq=2 ttl=63 time=2.169 ms
64 bytes from 172.16.101.3: seq=3 ttl=63 time=2.380 ms
64 bytes from 172.16.101.3: seq=4 ttl=63 time=1.823 ms
```

图 9-420

不出所料，cirros-vm1 和 cirros-vm3 能通信了。

接下来我们详细探究 router\_100\_101 是如何起作用的。

底层网络发生了什么变化

查看控制节点的网络结构发生了什么变化，如图 9-421 所示。

```
root@devstack-controller:~# ovs-vsctl show
c3f35829-cb18-41e5-b2e1-d378aaa9351d
    Bridge br-ex
        Port br-ex
            Interface br-ex
                type: internal
    Bridge br-int
        fail_mode: secure
        Port br-int
            Interface br-int
                type: internal
        Port "qvo1fb4cdbf-8f"
            tag: 1
            Interface "qvo1fb4cdbf-8f"
        Port "tap43567363-50"
            tag: 1
            Interface "tap43567363-50"
                type: internal
        Port "qr-d295b258-45"
            tag: 2
            Interface "qr-d295b258-45"
                type: internal
        Port "qr-2ffdb861-73"
            tag: 1
            Interface "qr-2ffdb861-73"
                type: internal
        Port "tap1820558c-0a"
```

图 9-421

br-int 上多了两个 port:

- qr-d295b258-45，从命名上可以推断该 interface 对应 router\_100\_101 的 interface (d295b258-4586)，是 subnet\_172\_16\_100\_0 的网关。



- qr-2ffdb861-73, 从命名上可以推断该 interface 对应 router\_100\_101 的 interface (2ffdb861-731c), 是 subnet\_172\_16\_101\_0 的网关。

与 Linux Bridge 实现方式一样, router\_100\_101 运行在自己的 namespace 中, 如图 9-422 所示。

```
root@devstack-controller:~# ip netns
qrouter-a81cc110-16f4-4d6c-89d2-8af91cec9714
qdhcp-390b9542-37bc-4210-bef6-040c89853ee3
qdhcp-324a77eb-5ab1-4eb2-896c-7303ef80828b
root@devstack-controller:~# ip netns exec qrouter-a81cc110-16f4-4d6c-89d2-8af91cec9714 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
32: qr-2ffdb861-73: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default
    link/ether fa:16:3e:9c:c3:88 brd ff:ff:ff:ff:ff:ff
    inet 172.16.100.1/24 brd 172.16.100.255 scope global qr-2ffdb861-73
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe9c:c388/64 scope link
        valid_lft forever preferred_lft forever
33: qr-d295b258-45: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default
    link/ether fa:16:3e:db:f9:8c brd ff:ff:ff:ff:ff:ff
    inet 172.16.101.1/24 brd 172.16.101.255 scope global qr-d295b258-45
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fedb:f98c/64 scope link
        valid_lft forever preferred_lft forever
root@devstack-controller:~#
```

图 9-422

如图 9-422 所示, qrouter-a81cc110-16f4-4d6c-89d2-8af91cec9714 为 router 的 namespace, 两个 Gateway IP 分别配置在 qr-2ffdb861-73 和 qr-d295b258-45 上。

当前网络结构如图 9-423 所示。

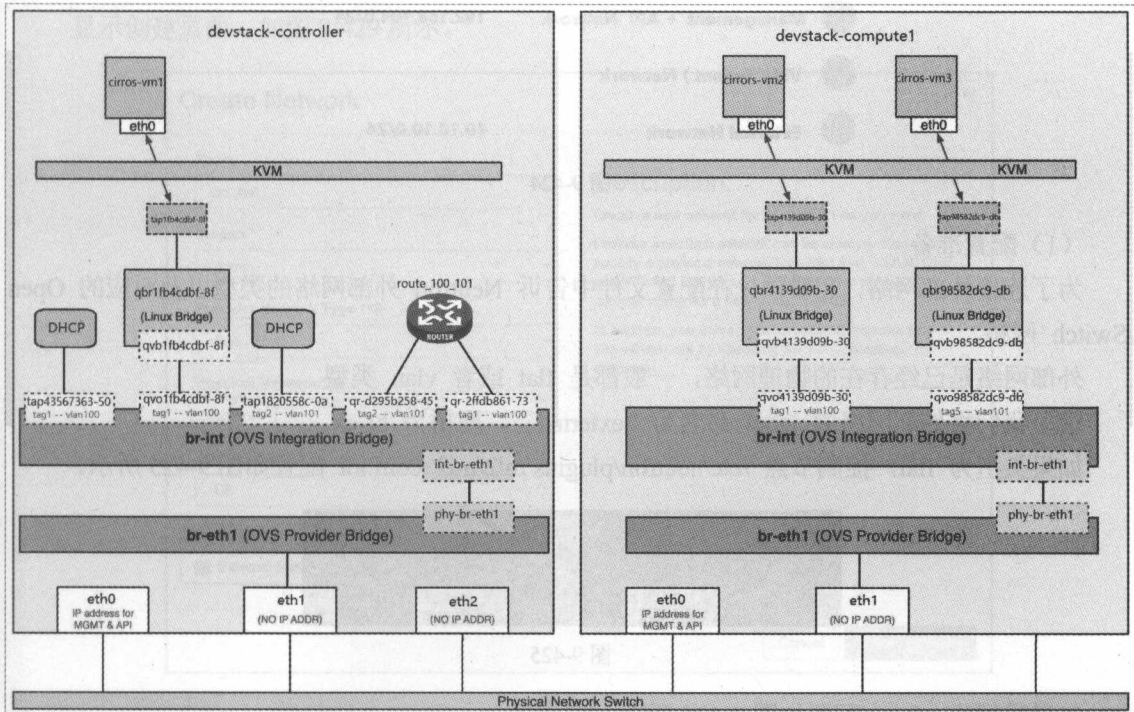


图 9-423

3. 访问外网

通过 router 可以实现位于不同 vlan 的 instance 之间的通信。

接下来要探讨的问题是 instance 如何与外部网络通信。这里的外部网络指的是租户网络以外的网络。租户网络是由 Neutron 创建和维护的网络。外部网络不由 Neutron 创建。如果是私有云，外部网络通常指的是公司 intranet；如果是公有云，外部网络通常指的是 internet。

具体到我们的实验网络环境：

- 计算节点和控制节点 eth1 提供的是租户网络，IP 段租户可以自由设置。
- 控制节点 eth2 连接的就是外部网络，IP 网段为 10.10.10.2/24。

如图 9-424 所示。

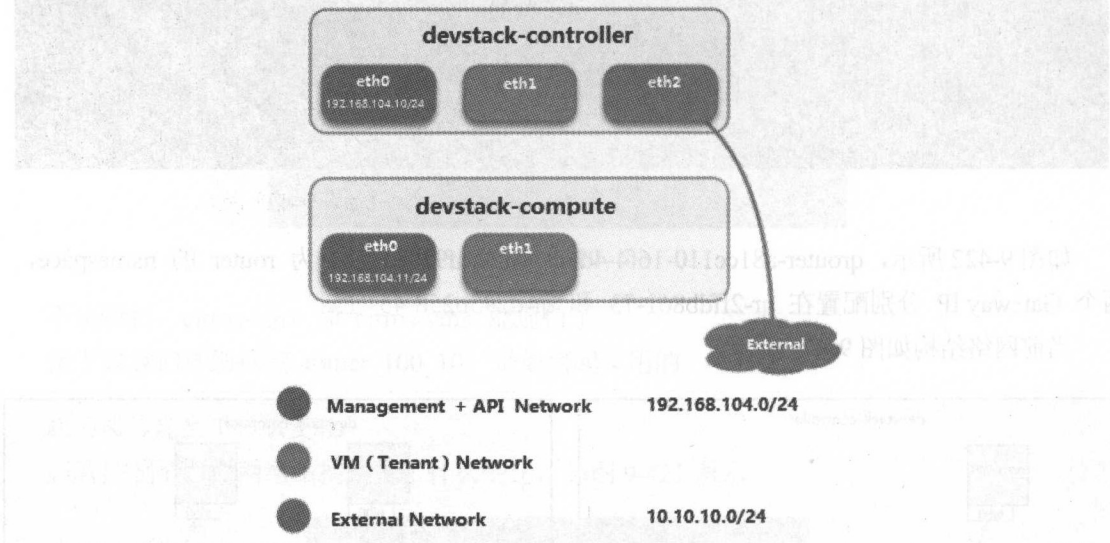


图 9-424

(1) 配置准备

为了连接外部网络，需要预先在配置文件中告诉 Neutron 外部网络的类型以及对应的 Open vSwitch 网桥。

外部网络是已经存在的物理网络，一般都是 flat 或者 vlan 类型。

这里我们将外部网络的 label 命名为“external”，网桥为 br-ex。

如果类型为 flat，控制节点 /etc/neutron/plugins/ml2/ml2\_conf.ini 配置如图 9-425 所示。

```
[ml2]
type = flat
flat_networks = external
bridge_mappings = default:br-eth1,external:br-ex
```

图 9-425

如果类型为 vlan，配置如图 9-426 所示。

```
networks:
  - name: ext_net
    type: flat
    segmentation_id: 100
    provider:
      type: flat
      segmentation_id: 100
    shared: true
    external: true
    default: true
    default: br-eth1,external:br-ex
```

图 9-426

在我们的网络环境中，外部网络是 flat 类型。

修改配置后，需要重启 neutron 的相关服务。

另外，我们需要提前准备好 br-ex，将 eth2 添加到 br-ex，如图 9-427 所示。

```
root@devstack-controller:~# ovs-vsctl add-br br-ex
ovs-vsctl: cannot create a bridge named br-ex because a bridge named br-ex already exists
root@devstack-controller:~#
root@devstack-controller:~# ovs-vsctl add-port br-ex eth2
root@devstack-controller:~#
```

图 9-427

br-ex 已经存在，我们只需要添加 eth2。

下一节我们演示如何创建外部网络 ext\_net。

## (2) 创建 ext\_net

进入菜单 Admin → Networks，单击“Create Network”，如图 9-428 所示。

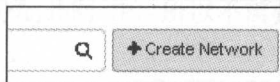


图 9-428

显示创建页面，如图 9-429 所示。

Create Network

Name

ext\_net

Project \*

admin

Provider Network Type \*

Flat

Physical Network \*

external

Admin State \*

UP

Shared

☒ External Network

Description:

Create a new network for any project as you need.

Provider specified network can be created. You can specify a physical network type (like Flat, VLAN, GRE, and VXLAN) and its segmentation\_id or physical network name for a new virtual network.

In addition, you can create an external network or a shared network by checking the corresponding checkbox.

Cancel

Create Network

图 9-429

在 Provider Network Type 下拉列表框中选择“Flat”。

在 Physical Network 下拉列表框中填写 “external”，与 ml2\_conf.ini 中 flat\_networks 的参数值保持一致。

选中 External Network 选择框。

单击 “Create Network”，ext\_net 创建成功，如图 9-430 所示。

单击 ext\_net 链接，进入 network 配置页面，目前还没有 subnet，单击 “Create Subnet”，如图 9-431 所示。

<input type="checkbox"/>	Project	Network Name	Subnets Associated
<input type="checkbox"/>	admin	ext_net	
<input type="checkbox"/>	admin	vlan101	subnet_172_16_101_0 172.16.101.0/24
<input type="checkbox"/>	admin	vlan100	subnet_172_16_100_0 172.16.100.0/24
Displaying 3 items			

图 9-430

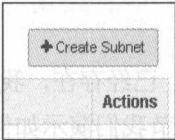


图 9-431

创建 subnet\_10\_10\_10\_0，IP 地址为 10.10.10.0/24，如图 9-432 所示。

Create Subnet

Subnet

Subnet Details

Subnet Name

subnet\_10\_10\_10\_0

Create a subnet associated with the network. Advanced configuration is available by clicking on the "Subnet Details" tab.

Network Address

10.10.10.0/24

IP Version

IPv4

Gateway IP

☐ Disable Gateway

« Back

Next »

图 9-432

这里 Gateway 我们使用默认地址 10.10.10.1。  
通常我们需要询问网络管理员外网 subnet 的 Gateway IP，然后填在这里。  
单击 “Next”，如图 9-433 所示。



Create Subnet

Subnet Details

☐ Enable DHCP

Specify additional attributes for the subnet.

Allocation Pools

DNS Name Servers

Host Routes

Back

Create

图 9-433

因为我们不会直接为 instance 分配外网 IP，所以不需要 enable DHCP。  
单击“Create”，如图 9-434 所示。

Subnets

Name	CIDR	IP Version	Gateway IP
subnet_10_10_10_0	10.10.10.0/24	IPv4	10.10.10.1

Displaying 1 item

图 9-434

subnet 创建成功，网关为 10.10.10.1。  
下面查看控制节点网络结构的变化，执行 ovs-vsctl show，如图 9-435 所示。

```
root@devstack-controller:~# ovs-vsctl show
c3f35829-cb18-41e5-b2e1-d378aaa9351d
  Bridge br-ex
    Port br-ex
      Interface br-ex
        type: internal
    Port phy-br-ex
      Interface phy-br-ex
        type: patch
        options: {peer=int-br-ex}
    Port "eth2"
      Interface "eth2"
  Bridge br-int
    fail_mode: secure
    Port br-int
      Interface br-int
        type: internal
    Port int-br-ex
      Interface int-br-ex
        type: patch
        options: {peer=phy-br-ex}
    Port "qr-2ffdb861-73"
```

图 9-435

如图 9-435 所示，br-ex 与 br-int 通过 patch port “phy-br-ex” 和 “int-br-ex” 连接。

(3) 将 ext\_net 连接到 router\_100\_101  
接下来需要将外网连接到 Neutron 的虚拟路由器，这样 instance 才能访问外网。  
单击菜单 Project → Network → Routers，进入 router 列表，如图 9-436 所示。

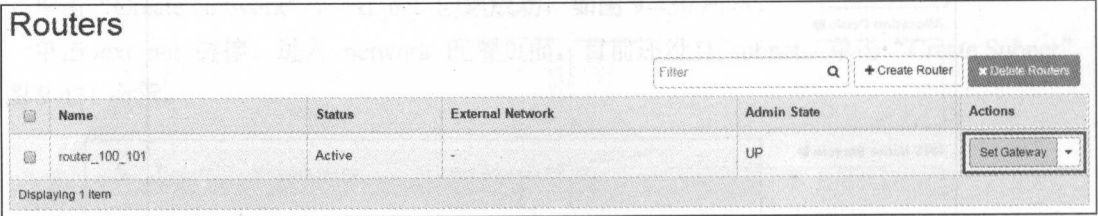


图 9-436

单击 router\_100\_101 的 “Set Gateway”，进入如图 9-437 所示。

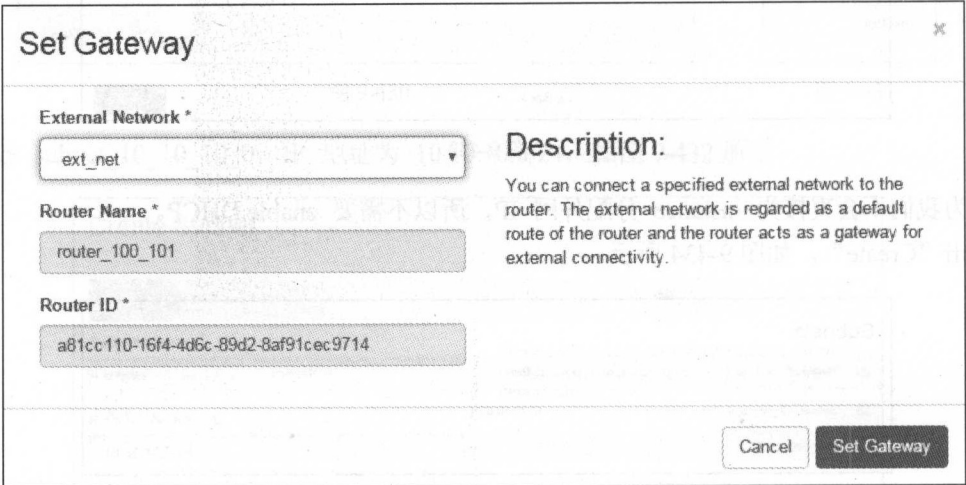


图 9-437

在 “External Network” 下拉列表中选择 ext\_net，单击 “Set Gateway”，如图 9-438 所示。

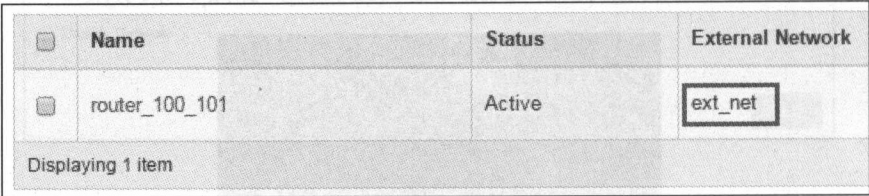


图 9-438

外网设置成功。我们需要看看 router 发生了什么变化。  
单击 “router\_100\_101” 链接，打开 “Interfaces” 标签页，如图 9-439 所示。

<input type="checkbox"/>	Name	Fixed IPs	Status	Type
<input type="checkbox"/>	(2ffdb861-731c)	172.16.100.1	Active	Internal Interface
<input type="checkbox"/>	(cf54d3ea-6a78)	10.10.10.2	Active	External Gateway
<input type="checkbox"/>	(d295b258-4586)	172.16.101.1	Active	Internal Interface
Displaying 3 Items				

图 9-439

router 多了一个新 interface, IP 为 10.10.10.2。

该 interface 用于连接外网 ext\_net, 对应 br-ex 的 port “qg-cf54d3ea-6a”, 如图 9-440 所示。

```

root@devstack-controller:~# ovs-vsctl show
c3f35829-cb18-41e5-b2e1-d378aaa9351d
    Bridge br-ex
        Port br-ex
            Interface br-ex
                type: internal
        Port "qg-cf54d3ea-6a"
            Interface "qg-cf54d3ea-6a"
                type: internal
        Port phy-br-ex
            Interface phy-br-ex
                type: patch
                options: {peer=int-br-ex}
        Port "eth2"
            Interface "eth2"

```

图 9-440

在 router 的 namespace 中查可以看到 qg-cf54d3ea-6a 已经配置了 IP 10.10.10.2, 如图 9-441 所示。

```

root@devstack-controller:~# ip netns exec qrouter-a81cc110-16f4-4d6c-89d2-8af91ce9714 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
8: qg-cf54d3ea-6a: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default
    link/ether fa:16:3e:96:49:9f brd ff:ff:ff:ff:ff:ff
    inet 10.10.10.2/24 brd 10.10.10.255 scope global qg-cf54d3ea-6a
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe96:499f/64 scope link
        valid_lft forever preferred_lft forever
12: qr-d295b258-45: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default
    link/ether fa:16:3e:db:f9:8c brd ff:ff:ff:ff:ff:ff
    inet 172.16.101.1/24 brd 172.16.101.255 scope global qr-d295b258-45
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fedb:f98c/64 scope link
        valid_lft forever preferred_lft forever
13: qr-2ffdb861-73: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default
    link/ether fa:16:3e:9c:c3:88 brd ff:ff:ff:ff:ff:ff
    inet 172.16.100.1/24 brd 172.16.100.255 scope global qr-2ffdb861-73
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe9c:c388/64 scope link
        valid_lft forever preferred_lft forever

```

图 9-441

router interface 的命名规则如下:

- 如果 interface 用于连接租户网络, 命名格式为 qr-xxx。
- 如果 interface 用于连接外部网络, 命名格式为 qg-xxx。

查看 router 的路由表信息，如图 9-442 所示。

```
root@devstack-controller:~# ip netns exec qrouter-a81cc110-16f4-4d6c-89d2-8af91cec9714 route
Kernel IP routing table
Destination      Gateway         Genmask         Flags Metric Ref    Use Iface
Default          10.10.10.1     0.0.0.0        UG    0     0          0 qg-cf54d3ea-6a
10.10.10.0       *              255.255.255.0  U    0     0          0 qg-cf54d3ea-6a
172.16.100.0     *              255.255.255.0  U    0     0          0 qr-2ffdb861-73
172.16.101.0     *              255.255.255.0  U    0     0          0 qr-d295b258-45
root@devstack-controller:~#
```

图 9-442

可以看到默认网关为 10.10.10.1。意味着对于访问 vlan100 和 vlan101 租户网络以外的所有流量，router\_100\_101 都将转发给 ext\_net 的网关 10.10.10.1。

现在 router\_100\_101 已经同时连接了 vlan100、vlan101 和 ext\_net 三个网络，如图 9-443 所示。

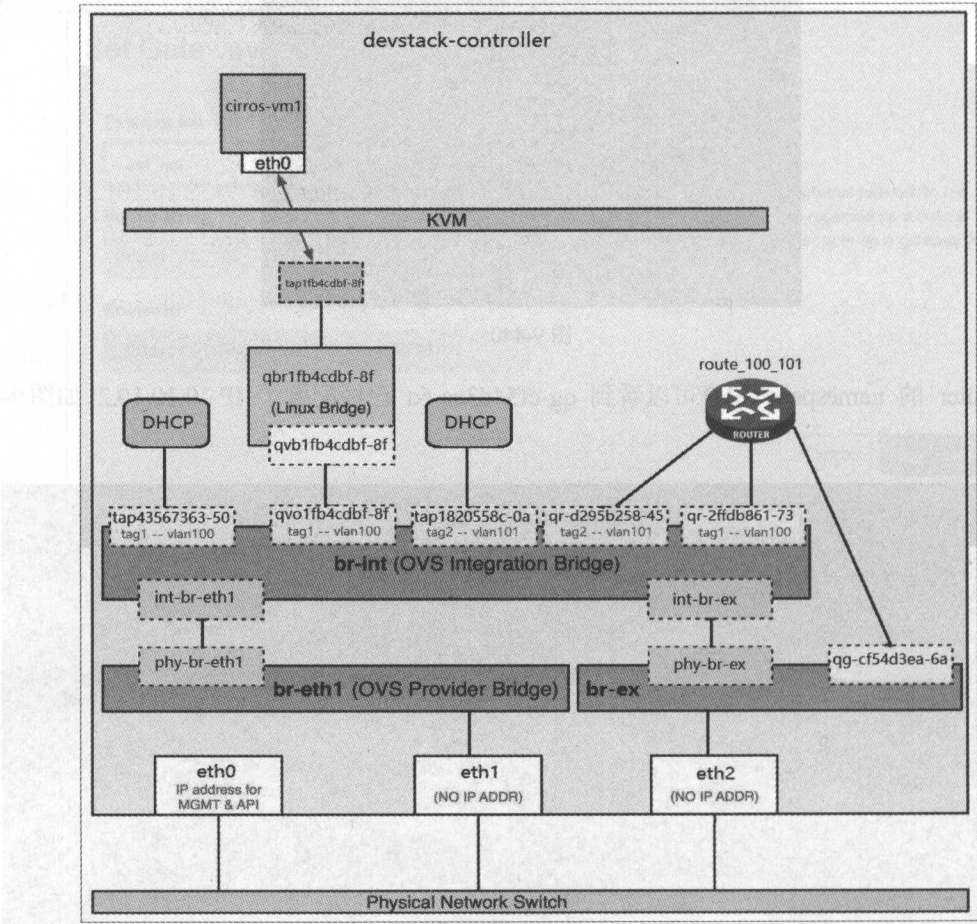


图 9-443

我们在 cirros-vm3 上测试一下，如图 9-444 所示。



```

$ hostname
cirros-vm3
$ ping 10.10.10.1
PING 10.10.10.1 (10.10.10.1): 56 data bytes
64 bytes from 10.10.10.1: seq=0 ttl=254 time=4.238 ms
64 bytes from 10.10.10.1: seq=1 ttl=254 time=3.505 ms
64 bytes from 10.10.10.1: seq=2 ttl=254 time=1.429 ms
64 bytes from 10.10.10.1: seq=3 ttl=254 time=1.653 ms
64 bytes from 10.10.10.1: seq=4 ttl=254 time=8.996 ms

```

图 9-444

cirros-vm3 位于计算节点，现在已经可以 Ping 到 ext\_net 网关 10.10.10.1 了。

通过 traceroute 查看一下 cirros-vm3 到 10.10.10.1 的路径，如图 9-445 所示。

```

$ traceroute 10.10.10.1
traceroute to 10.10.10.1 (10.10.10.1), 30 hops max, 46 byte packets
 1  host-172-16-101-1.openstacklocal (172.16.101.1)  2.287 ms  0.740 ms  0.836 ms
 2  10.10.10.1 (10.10.10.1)  2.816 ms  1.414 ms  1.232 ms

```

图 9-445

数据包经过两跳到达 10.10.10.1 网关。

- (1) 数据包首先发送到 router\_100\_101 连接 vlan101 的 interface (172.16.101.1)。
- (2) 然后通过连接 ext\_net 的 interface (10.10.10.2) 转发出去，最后到达 10.10.10.1。

当数据包从 router 连接外网的接口 qg-cf54d3ea-6a 发出的时候，会做一次 Source NAT，将包的源地址修改为 router 的接口地址 10.10.10.2，这样就能够保证目的端能够将应答的包发回给 router，然后再转发回源端 instance。

有关 Source NAT 的详细分析可以参考 Linux Bridge 中 router 的相关章节。

#### 4. floating IP

通过 SNAT 使得 instance 能够直接访问外网，但外网还不能直接访问 instance。直接访问 instance 指的是通信连接由外网发起，例如从外网 SSH instance。如果需要从外网直接访问 instance，可以利用 floating IP。

Open vSwitch driver 环境中，floating IP 的实现与 Linux Bridge driver 完全一样：都是通过 router 提供网关的外网 interface 上配置 iptables NAT 规则实现。

有关 floating IP 的详细分析可以参考 Linux Bridge 中 floating IP 的相关章节。

### 9.5.9 vxlan network

Open vSwitch 支持 VXLAN 和 GRE 这两种 overlay network。因为 OpenStack 对于 VXLAN 与 GRE 配置和实现差别不大，这里只讨论如何实施 VXLAN。

有关 VXLAN 的原理和概念请参考 Linux Bridge 中 VXLAN 相关章节，这里不再赘述。

#### 1. 在 ML2 配置中 enable vxlan network

在 /etc/neutron/plugins/ml2/ml2\_conf.ini 设置 vxlan network 相关参数，如图 9-446 所示。

```
[ml2]
tenant_network_types = vxlan
extension_drivers = port_security
type_drivers = local,flat,vlan,gre,vxlan
mechanism_drivers = openvswitch,l2population
```

图 9-446

指定普通用户创建的网络类型为 vxlan，同时也 enable l2population mechanism driver。然后指定 vxlan 的范围，如图 9-447 所示。

```
[ml2_type_vxlan]
vni_ranges = 1001:2000
```

图 9-447

上面配置定义了 vxlan vni 的范围是 1001~2000。这个范围是针对普通用户在自己的租户里创建 vxlan network 的范围。因为普通用户创建 network 时不能指定 vni，Neutron 会按顺序自动从这个范围中取值。

对于 admin 则没有 vni 范围的限制，admin 可以创建 vni 范围为 1~16777216 的 vxlan network。

在 [agent] 中配置启用 vxlan 和 l2\_population，如图 9-448 所示。

```
[agent]
tunnel_types = vxlan
l2_population = True
```

图 9-448

最后在 [ovs] 中配置 VTEP，如图 9-449 所示。

```
[ovs]
bridge_mappings =
tunnel_bridge = br-tun
local_ip = 166.66.16.10
```

图 9-449

vxlan tunnel 对应的网桥为 br-tun。local\_ip 指定 VTEP 的 IP 地址。

devstack\_controller 使用 166.66.16.10，此 IP 配置在网卡 eth1 上。

devstack\_compute01 则使用 166.66.16.11，此 IP 配置在网卡 eth1 上，如图 9-450 所示。

```
[ovs]
bridge_mappings =
tunnel_bridge = br-tun
local_ip = 166.66.16.11
```

图 9-450

## 2. 初始网络结构

Neutron 服务重启后，通过 ovs-vsctl show 查看网络配置，如图 9-451 所示。

```

root@devstack-controller:~# ovs-vsctl show
c3f35829-cb18-41e5-b2e1-d378aaa9351d
    Bridge br-int
        fail_mode: secure
        Port br-int
            Interface br-int
                type: internal
            Port patch-tun
                Interface patch-tun
                    type: patch
                    options: {peer=patch-int}
    Bridge br-tun
        fail_mode: secure
        Port br-tun
            Interface br-tun
                type: internal
            Port patch-int
                Interface patch-int
                    type: patch
                    options: {peer=patch-tun}
    ovs_version: "2.0.2"

```

图 9-451

br-int 与 br-tun 通过 patch port “patch-tun” 和 “br-tun” 连接。  
目前网络结构如图 9-452 所示。

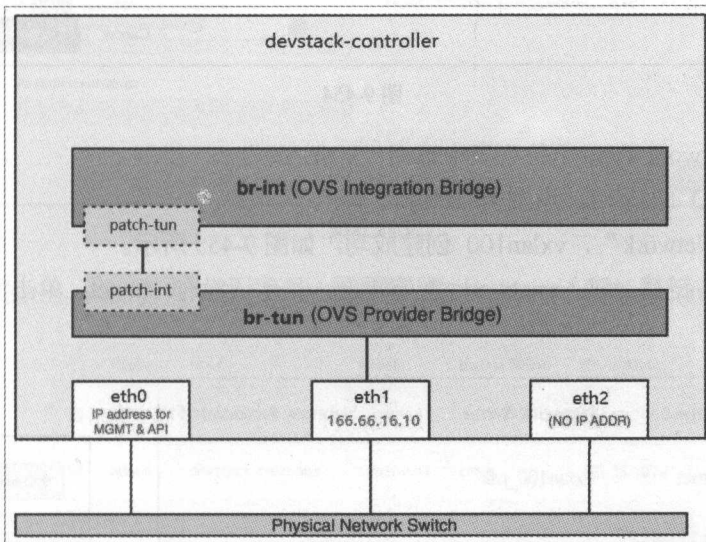


图 9-452

### 3. 创建 vxlan network“vxlan100\_net”

打开菜单 Admin → Networks, 单击 “Create Network”, 如图 9-453 所示。

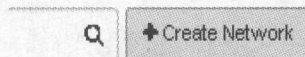


图 9-453

显示创建页面, 如图 9-454 所示。

Create Network

Name

vxlan100\_net

Project \*

admin

Provider Network Type \* ?

VXLAN

Segmentation ID \* ?

100

Admin State \*

UP

☐ Shared

☐ External Network

Description:

Create a new network for any project as you need.  
Provider specified network can be created. You can specify a physical network type (like Flat, VLAN, GRE, and VXLAN) and its segmentation\_id or physical network name for a new virtual network.  
In addition, you can create an external network or a shared network by checking the corresponding checkbox.

Cancel

Create Network

图 9-454

在 Provider Network Type 下拉列表中选择“VXLAN”。

Segmentation ID 即 VNI，设置为 100。

单击“Create Network”，vxlan100 创建成功，如图 9-455 所示。

单击 vxlan100 链接，进入 network 配置页面，目前还没有 subnet，单击“Create Subnet”，如图 9-456 所示。

<input type="checkbox"/>	Project	Network Name	Subnets Associated
<input type="checkbox"/>	admin	vxlan100_net	
Displaying 1 item			

图 9-455

+ Create Subnet

Actions

图 9-456

创建 subnet\_172\_16\_100\_0，IP 地址为 172.16.100.0/24，如图 9-457、图 9-458 所示。

Create Subnet

Subnet

Subnet Details

Subnet Name

subnet\_172\_16\_100\_0

Network Address ?

172.16.100.0/24

图 9-457



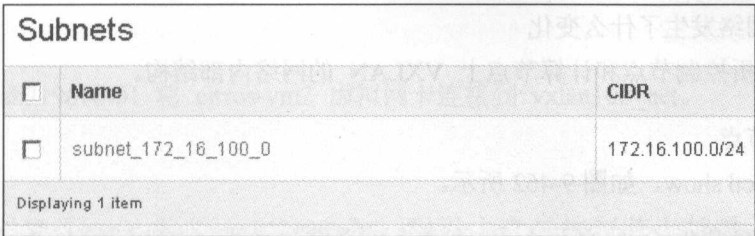


图 9-458

(1) 将 instance 连接到 vxlan100\_net  
launch 新的 instance “cirros-vm1”，“cirros-vm2” 网络选择 vxlan100，如图 9-459 所示。

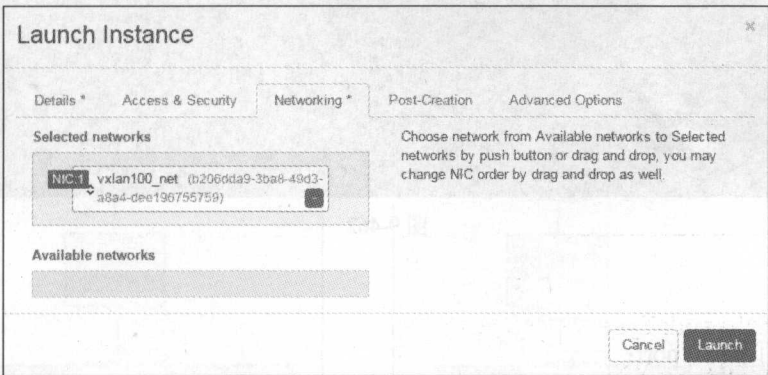


图 9-459

cirros-vm1, cirros-vm2 分别部署到控制节点和计算节点，IP 如图 9-460 所示。

<input type="checkbox"/>	Project	Host	Name	Image Name	IP Address
<input type="checkbox"/>	admin	devstack-compute1	cirros-vm2	cirros	172.16.100.6
<input type="checkbox"/>	admin	devstack-controller	cirros-vm1	cirros	172.16.100.5

Displaying 2 items

图 9-460

测试 cirros-vm1 和 cirros-vm2 的连通性，如图 9-461 所示。

```
$ hostname
cirros-vm1
$
$ ping 172.16.100.6
PING 172.16.100.6 (172.16.100.6): 56 data bytes
64 bytes from 172.16.100.6: seq=0 ttl=64 time=3.823 ms
64 bytes from 172.16.100.6: seq=1 ttl=64 time=2.649 ms
64 bytes from 172.16.100.6: seq=2 ttl=64 time=2.425 ms
64 bytes from 172.16.100.6: seq=3 ttl=64 time=2.379 ms
64 bytes from 172.16.100.6: seq=4 ttl=64 time=2.172 ms
```

图 9-461

与我们预期相同，cirros-vm1 能 Ping 通 cirros-vm2。  
接下来我们详细分析 Open vSwitch 是如何实现 VXLAN 的。

(2) 底层网络发生了什么变化

本节详细分析控制节点和计算节点上 VXLAN 的网络内部结构。

(3) 控制节点

执行 `ovs-vsctl show`，如图 9-462 所示。

```
root@devstack-controller:~# ovs-vsctl show
c3f35829-cb18-41e3-b2e1-d378aaa9351d
  Bridge br-int
    fail_mode: secure
    Port br-int
      interface br-int
      type: internal
    Port "tap0d4cb13a-7a"
      tag: 1
      interface "tap0d4cb13a-7a"
      type: internal
    Port patch-tun
      interface patch-tun
      type: patch
      options: {peer=patch-int}
    Port "qvoa2ac3b9a-24"
      tag: 1
      interface "qvoa2ac3b9a-24"
  Bridge br-tun
    fail_mode: secure
    Port br-tun
      interface br-tun
      type: internal
    Port patch-int
      interface patch-int
      type: patch
      options: {peer=patch-tun}
    Port "vxlan-a642100b"
      interface "vxlan-a642100b"
      type: vxlan
      options: {df_default="true", in_key=flow, local_ip="166.66.16.10", out_key=flow, remote_ip="166.66.16.11"}
  ovs_version: "2.0.2"
```

图 9-462

● br-int

br-int 连接了如下 port:

tap0d4cb13a-7a 是 vxlan100\_net 的 DHCP 服务对应的 interface。

qvoa2ac3b9a-24 将 cirros-vm1 虚拟网卡连接到 vxlan100\_net。

● br-tun

br-tun 上创建了一个特殊的 port “vxlan-a642100b”，它是 VXLAN 的隧道端点，指定了本地（devstack-controller）节点和远端（devstack-compute1）节点 VTEP 的 IP。

(4) 计算节点

执行 `ovs-vsctl show`，如图 9-463 所示。

```
root@devstack-compute1:~# ovs-vsctl show
c843cf2c-9c70-4e2e-90c9-ce58a2997fc6
  Bridge br-int
    fail_mode: secure
    Port patch-tun
      interface patch-tun
      type: patch
      options: {peer=patch-int}
    Port "qvoab219616-01"
      tag: 2
      interface "qvoab219616-01"
    Port br-int
      interface br-int
      type: internal
  Bridge br-tun
    fail_mode: secure
    Port patch-int
      interface patch-int
      type: patch
      options: {peer=patch-tun}
    Port "vxlan-a642100a"
      interface "vxlan-a642100a"
      type: vxlan
      options: {df_default="true", in_key=flow, local_ip="166.66.16.11", out_key=flow, remote_ip="166.66.16.10"}
    Port br-tun
      interface br-tun
      type: internal
  ovs_version: "2.0.2"
```

图 9-463

- br-int

br-int 上 qvoab219616-01 将 cirros-vm2 虚拟网卡连接到 vxlan100\_net。

- br-tun

br-tun 上也创建了 port “vxlan-a642100b”，配置内容与控制节点相对，指定了本地（devstack-compute1）节点和远端（devstack-controller）节点 VTEP 的 IP。

当前网络结构如图 9-464 所示。

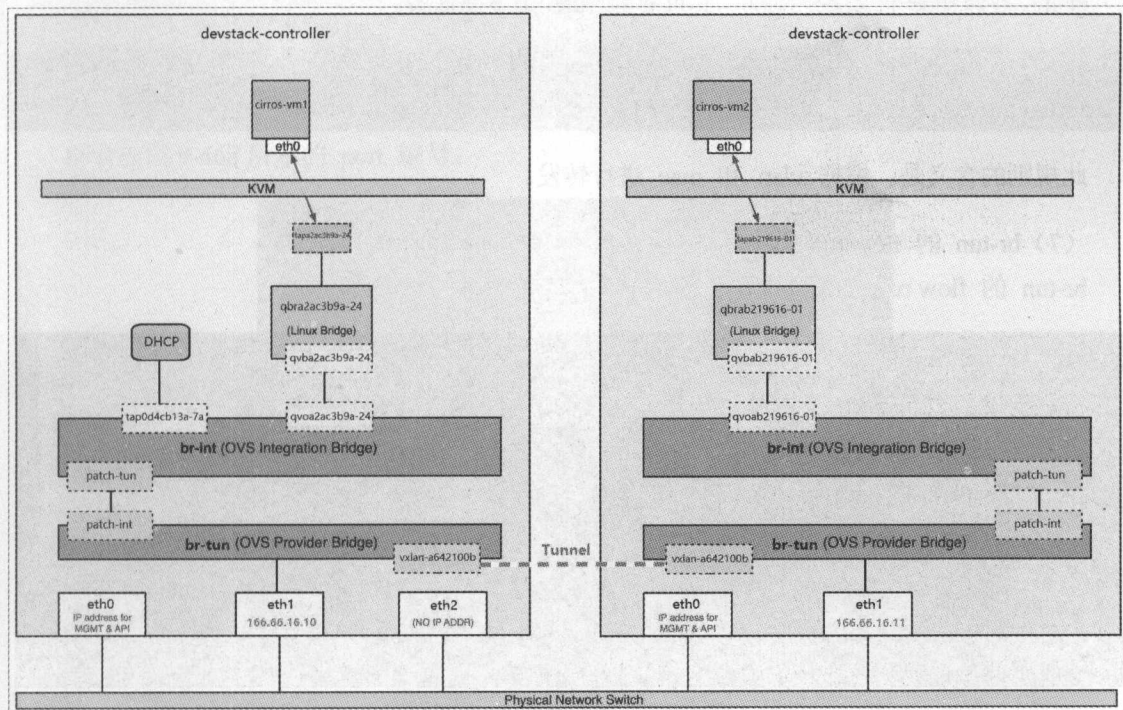


图 9-464

需要特别注意的是：无论存在多少个 VXLAN，devstack-controller 与 devstack-compute1 之间所有的数据都只通过 “vxlan-a642100b” 这对 port 上建立的隧道传输。

### (5) flow rule 分析

本节分析 Open vSwitch 如何通过 flow rule 实现 VXLAN 通信。

下面我们分析控制节点上的 flow rule，计算节点类似。

### (6) br-int 的 flow rule

如图 9-465 所示，br-int 的 rule 看上去虽然多，其实逻辑很简单。

```
root@devstack-controller:~# ovs-ofctl dump-flows br-int
NXST_FLOW reply (xid=0x4):
  cookie=0xaa0e760a7848ec3, duration=52680.992s, table=0, n_packets=0, n_bytes=0, idle_age=52680, priority=10,icmp6,in_port=3,icmp_type=136 actions=resubmit(,24)
  cookie=0xaa0e760a7848ec3, duration=52680.983s, table=0, n_packets=15, n_bytes=630, idle_age=9465, priority=10,arp,in_port=3 actions=resubmit(,24)
  cookie=0xaa0e760a7848ec3, duration=52798.625s, table=0, n_packets=143, n_bytes=14594, idle_age=9415, priority=0 actions=NORMAL
  cookie=0xaa0e760a7848ec3, duration=52798.608s, table=23, n_packets=0, n_bytes=0, idle_age=52798, priority=0 actions=drop
  cookie=0xaa0e760a7848ec3, duration=52680.997s, table=24, n_packets=0, n_bytes=0, idle_age=52680, priority=2,icmp6,in_port=3,icmp_type=136,nd_target=fe80::f816:3eff:f
eed:d309 actions=NORMAL
  cookie=0xaa0e760a7848ec3, duration=52680.988s, table=24, n_packets=15, n_bytes=630, idle_age=9465, priority=2,arp,in_port=3,arp_spa=172.16.100.5 actions=NORMAL
  cookie=0xaa0e760a7848ec3, duration=52798.588s, table=24, n_packets=0, n_bytes=0, idle_age=52798, priority=0 actions=drop
```

图 9-465

br-int 被当作一个二层交换机，其重要的 rule 是下面这条：

```
cookie=0xaa0e760a7848ec3, duration=52798.625s, table=0, n_packets=143,
n_bytes=14594, idle_age=9415, priority=0 actions=NORMAL
```

此规则的含义是：根据 vlan 和 mac 进行转发。

(7) br-tun 的 flow rule

br-tun 的 flow rule 如图 9-466 所示。

```
root@devstack-controller:~# ovs-ofctl dump-flows br-tun
NXST_FLOW reply (xid=0x4):
  cookie=0xaa0e760a7848ec3, duration=76707.867s, table=0, n_packets=70, n_bytes=6600, idle_age=33324, hard_age=65534, priority=1,in_port=1 actions=resubmit(,2)
  cookie=0xaa0e760a7848ec3, duration=76543.287s, table=0, n_packets=56, n_bytes=4948, idle_age=33324, hard_age=65534, priority=1,in_port=2 actions=resubmit(,4)
  cookie=0xaa0e760a7848ec3, duration=76707.867s, table=0, n_packets=0, n_bytes=0, idle_age=65534, hard_age=65534, priority=0 actions=drop
  cookie=0xaa0e760a7848ec3, duration=76707.866s, table=2, n_packets=28, n_bytes=3180, idle_age=33324, hard_age=65534, priority=0,d1_dst=00:00:00:00:00:01:00:
00:00:00 actions=resubmit(,20)
  cookie=0xaa0e760a7848ec3, duration=76707.866s, table=2, n_packets=42, n_bytes=3420, idle_age=33379, hard_age=65534, priority=0,d1_dst=01:00:00:00:00:00:01:00:
00:00:00:00 actions=resubmit(,22)
  cookie=0xaa0e760a7848ec3, duration=76707.866s, table=3, n_packets=0, n_bytes=0, idle_age=65534, hard_age=65534, priority=0 actions=drop
  cookie=0xaa0e760a7848ec3, duration=76647.039s, table=4, n_packets=56, n_bytes=4948, idle_age=33324, hard_age=65534, priority=1,tun_id=0x64 actions=mod_vlan_vl
d:1, resubmit(,10)
  cookie=0xaa0e760a7848ec3, duration=76707.866s, table=4, n_packets=0, n_bytes=0, idle_age=65534, hard_age=65534, priority=0 actions=drop
  cookie=0xaa0e760a7848ec3, duration=76707.865s, table=6, n_packets=0, n_bytes=0, idle_age=65534, hard_age=65534, priority=0 actions=drop
  cookie=0xaa0e760a7848ec3, duration=76707.865s, table=10, n_packets=56, n_bytes=4948, idle_age=33324, hard_age=65534, priority=1 actions=learn(table=20,hard_r
egion=300,priority=1,cookie=0xaa0e760a7848ec3,NXM_OF_VLAN_TCI[0..11],NXM_OF_ETH_DST[]->NXM_OF_ETH_SRC[],load:0->NXM_OF_VLAN_TCI[],load:NXM_NX_TUN_ID[]->NXM_NX_T
UN_ID[],output:NXM_OF_IN_PORT[],output:1
  cookie=0xaa0e760a7848ec3, duration=76707.865s, table=20, n_packets=0, n_bytes=0, idle_age=65534, hard_age=65534, priority=0 actions=resubmit(,22)
  cookie=0xaa0e760a7848ec3, duration=76543.287s, table=20, n_packets=28, n_bytes=3180, idle_age=33324, hard_age=65534, priority=2,d1_vlan=1,d1_dst=fa:16:3e:fd:8
a:ed actions=strip_vlan,set_tunnel:0x64,output:2
  cookie=0xaa0e760a7848ec3, duration=76543.282s, table=22, n_packets=2, n_bytes=84, idle_age=33379, hard_age=65534, d1_vlan=1 actions=strip_vlan,set_tunnel:0x64
,output:2
  cookie=0xaa0e760a7848ec3, duration=76707.82s, table=22, n_packets=40, n_bytes=3336, idle_age=65534, hard_age=65534, priority=0 actions=drop
```

图 9-466

这些才是真正处理 VXLAN 数据包的 rule，流程如图 9-467 所示。

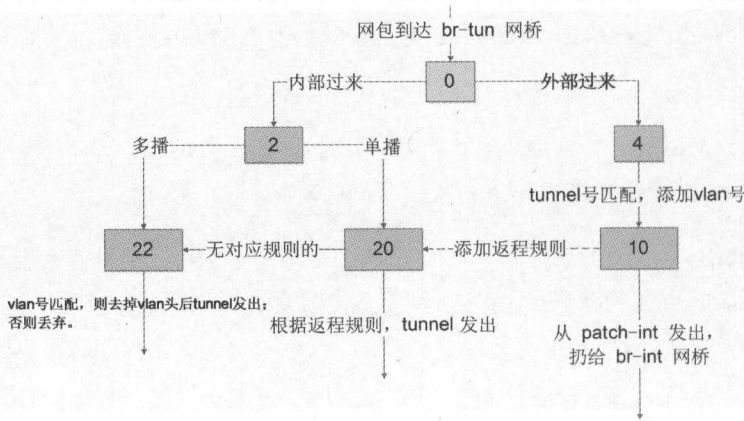


图 9-467



上图各方块中的数字对应 rule 中 table 的序号, 比如编号为 0 的方块对应下面三条 rule。

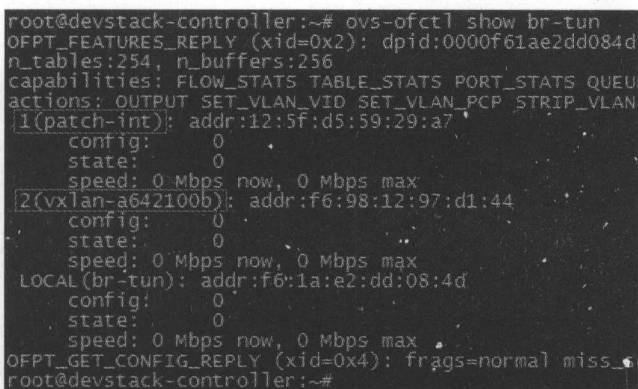
#### ● table 0

```
cookie=0xaa0e760a7848ec3, duration=76707.867s, table=0, n_packets=70,
n_bytes=6600, idle_age=33324, hard_age=65534, priority=1, in_port=1
actions=resubmit(,2)

cookie=0xaa0e760a7848ec3, duration=76543.287s, table=0, n_packets=56,
n_bytes=4948, idle_age=33324, hard_age=65534, priority=1, in_port=2
actions=resubmit(,4)

cookie=0xaa0e760a7848ec3, duration=76707.867s, table=0, n_packets=0,
n_bytes=0, idle_age=65534, hard_age=65534, priority=0 actions=drop
```

结合如图 9-468 所示的 port 编号。



```
root@devstack-controller:~# ovs-ofctl show br-tun
OFPT_FEATURES_REPLY (xid=0x2): dpid:0000f61ae2dd084d
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS
actions: OUTPUT SET_VLAN_VID SET_VLAN_PCP STRIP_VLAN
1(patch-int): addr:12:5f:d5:59:29:a7
  config: 0
  state: 0
  speed: 0 Mbps now, 0 Mbps max
2(vxlan-a642100b): addr:f6:98:12:97:d1:44
  config: 0
  state: 0
  speed: 0 Mbps now, 0 Mbps max
LOCAL(br-tun): addr:f6:1a:e2:dd:08:4d
  config: 0
  state: 0
  speed: 0 Mbps now, 0 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss
root@devstack-controller:~#
```

图 9-468

table 0 flow rule 的含义为:

- (1) 从 port 1 (patch-int) 进来的包, 扔给 table 2 处理: actions=resubmit(,2)。
- (2) 从 port 2 (vxlan-a642100b) 进来的包, 扔给 table 4 处理: actions=resubmit(,4)。

即第一条 rule 处理来自内部 br-int (这上面挂载着所有的网络服务, 包括路由、DHCP 等) 的数据; 第二条 rule 处理来自外部 VXLAN 隧道的数据。

#### ● table 4

```
cookie=0xaa0e760a7848ec3, duration=76647.039s, table=4, n_packets=56,
n_bytes=4948, idle_age=33324, hard_age=65534, priority=1, tun_id=0x64
actions=mod_vlan_vid:1,resubmit(,10)
```

table 4 flow rule 的含义为: 如果数据包的 VXLAN tunnel ID 为 100 (tun\_id=0x64), action 是添加内部 VLAN ID 1 (tag=1), 然后扔给 table 10 去学习。

### ● table 10

```
cookie=0xaaa0e760a7848ec3, duration=76707.865s, table=10, n_packets=56,
n_bytes=4948, idle_age=33324, hard_age=65534, priority=1
actions=learn(table=20,hard_timeout=300,priority=1,cookie=0xaaa0e760a784
8ec3,NXMOFVLANTCI[0..11],NXMOFETHDST[]=NXMOFETHSRC[],load:0->NXMOFVLANTC
I[],load:NXMNXTUNID[]->NXMNXTUNID[],output:NXMOFINPORT[]),output:1
```

table 10 flow rule 的含义为：学习外部（从 tunnel）进来的包，往 table 20 中添加对返程包的正常转发规则，然后从 port 1（patch-int）扔给 br-int。

rule 中下面的内容为学习规则，这里就不详细讨论了。

```
NXMOFVLANTCI[0..11],NXMOFETHDST[]=NXMOFETHSRC[],load:0->NXMOFVLANTCI[
],load:NXMNXTUNID[]->NXMNXTUNID[],output:NXMOFIN_PORT[]
```

### ● table 2

```
cookie=0xaaa0e760a7848ec3, duration=76707.866s, table=2, n_packets=28,
n_bytes=3180, idle_age=33324, hard_age=65534,
priority=0,d1_dst=00:00:00:00:00:00/01:00:00:00:00:00
actions=resubmit(,20)

cookie=0xaaa0e760a7848ec3, duration=76707.866s, table=2, n_packets=42,
n_bytes=3420, idle_age=33379, hard_age=65534,
priority=0,d1_dst=01:00:00:00:00:00/01:00:00:00:00:00
actions=resubmit(,22)
```

table 2 flow rule 的含义为：

- (1) br-int 发过来数据如果是单播包，扔给 table 20 处理：resubmit(,20)。
- (2) br-int 发过来数据如果是多播或广播包，扔 table 22 处理：resubmit(,22)。

### ● table 20

```
cookie=0xaaa0e760a7848ec3, duration=76543.287s, table=20, n_packets=28,
n_bytes=3180, idle_age=33324, hard_age=65534,
priority=2,d1_vlan=1,d1_dst=fa:16:3e:fd:8a:ed
actions=strip_vlan,set_tunnel:0x64,output:2

cookie=0xaaa0e760a7848ec3, duration=76707.865s, table=20, n_packets=0,
n_bytes=0, idle_age=65534, hard_age=65534, priority=0 actions=resubmit(,22)
```

table 20 flow rule 的含义为：

- (1) 第一条规则就是 table 10 学习来的结果。内部 VLAN 号为 1（tag=1），目标 MAC 是

fa:16:3e:fd:8a:ed (virros-vm2) 的数据包, 即发送给 virros-vm2 的包, action 是去掉 VLAN 号, 添加 VXLAN tunnel ID 100(十六进制 0x64), 并从 port 2 (tunnel 端口 vxlan-a642100b) 发出。

(2) 对于没学习到规则的数据包, 则扔给 table 22 处理。

#### ● table 22

```
cookie=0xaaa0e760a7848ec3, duration=76543.282s, table=22, n_packets=2,
n_bytes=84, idle_age=33379, hard_age=65534, dl_vlan=1
actions=strip_vlan,set_tunnel:0x64,output:2
cookie=0xaaa0e760a7848ec3, duration=76707.82s, table=22, n_packets=40,
n_bytes=3336, idle_age=65534, hard_age=65534, priority=0 actions=drop
```

table 22 flow rule 的含义为: 如果数据包的内部 VLAN 号为 1(tag=1), action 是去掉 VLAN 号, 添加 VXLAN tunnel ID 100 (十六进制 0x64), 并从 port 2 (tunnel 端口 vxlan-a642100b) 发出。

#### 4. VXLAN 的路由和 floating IP 支持

对于多 VXLAN 之间的 routing 以及 floating IP, 实现方式与 vlan 非常类似, 这里不再赘述, 请参看前面 vlan 相关章节。

## 9.6 总结

本章重点讨论 Neutron 的架构, 并通过分析 Linux Bridge 和 Open vSwitch 两个 mechanism driver 的技术细节, 实践了 local、flat、vlan、vxlan 四种网络类型, 同时也讨论了 routing 以及 floating IP 的实现细节。

Linux Bridge 和 Open vSwitch 都支持 Secure Group、Firewall as a Service、Load Balancing as a Service 等高级功能, 其实现方式也大致相同。

通过本章的学习, 大家应该能够掌握 Neutron 的理论知识并应用到实践部署中。

# 写在最后

作为 OpenStack 的基础教程，我们已经到了该收尾的地方。

本教程涵盖了 OpenStack 最最核心的模块：Keystone、Nova、Glance、Cinder 和 Neutron，通过大量的实验探讨了 OpenStack 的运行机制。

这个教程的目标是使读者能够掌握实施和管理 OpenStack 的必需技能，能够真正将 OpenStack 用起来。

为了达到这个目标，每一章都设计了大量的实践操作环节，通过截图和日志帮助读者理解各个技术要点，同时为读者自己实践 OpenStack 提供详尽的参考。

本教程对读者应该会有两个作用：

- 初学者可以按照章节顺序系统地学习 OpenStack，并通过教程中的实验掌握 OpenStack 的部署知识。
- 有经验的运维人员可以将本教程当做参考材料，在实际工作中有针对性地查看相关知识点。

OpenStack 是一个功能强大的云操作系统，除了本教程重点讨论的核心组件，还有数量众多的外围组件不断地丰富着 OpenStack 的功能。

所谓磨刀不费砍柴工，如果读者们能够通过本教程打下坚实基础，能够从容地运维 OpenStack 核心组件，再在此基础上学习和使用其他高级组件将一定会事半功倍。

最后祝大家使用 OpenStack 愉快！



## 作者简介

CloudMan，云计算技术专家，就职于国际知名IT企业，负责OpenStack相关项目的规划和实施。十多年一直专注IT前沿技术的钻研与实践，目前重点研究OpenStack、容器技术栈、DevOps等技术领域。

Internet of Things

清华大学出版社数字出版网站

WQBook 书文局泉

www.wqbook.com

Big Data

Cloud Computing

ISBN 978-7-302-45531-8



9 787302 455318 >

定价：89.00元